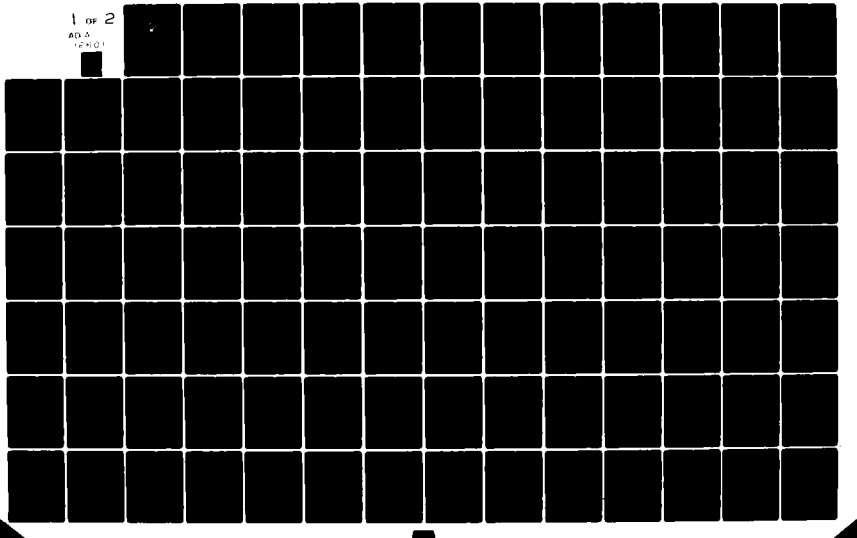
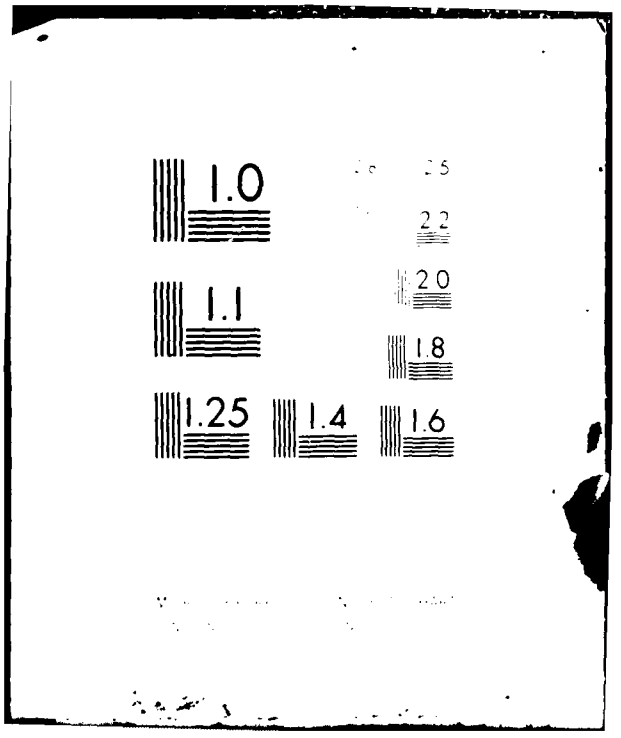
AD-A112 801 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 14/1
DYNAMIC PLANNING AND CONTROL OF SOFTWARE MAINTENANCE: A FISCAL --ETC(U)
DEC 81 J F GREEN, B F SELBY

UNCLASSIFIED NL

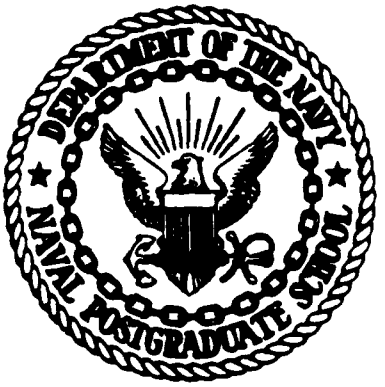
1 OF 2
AD A
112801




1.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

DYNAMIC PLANNING AND CONTROL
OF SOFTWARE MAINTENANCE:
A FISCAL APPROACH

by

James F. Green

Brenda F. Selby

Thesis Advisors: Lyle A. Cox Jr.

Danial C. Boger

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DYNAMIC PLANNING AND CONTROL OF SOFTWARE MAINTENANCE: A FISCAL APPROACH		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1981 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James F. Green Brenda F. Selby		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1981 13. NUMBER OF PAGES 128
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Maintenance
Software Life-cycle
Software Evolution
Software Macroestimation
Software Microestimation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

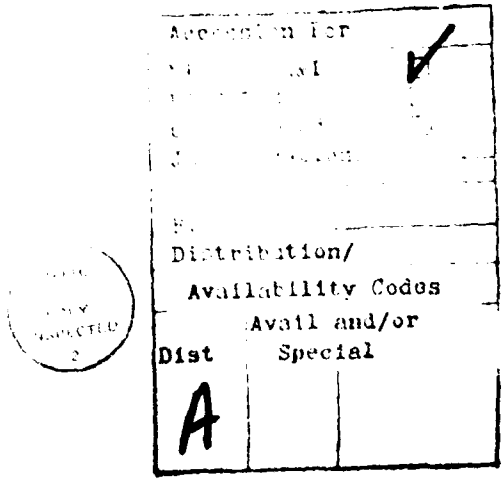
Until recently, much of the budget planning for software systems has been primarily targeted at costs incurred during the development phase. However, with increasing software system life span and complexity, maintenance costs have become a more prevalent concern. As a result of necessary corrections for design errors and evolutionary maintenance, post-delivery investment in software systems now requires a greater proportional share of the life-cycle costs. In this research, various methodologies and system factors relating to software cost accounting are reviewed with the intent of developing a cost

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

control model for arriving at a well-structured view for the management of the maintenance phase of the software life-cycle. The model proposed embodies a planning concept for establishing a maintenance strategy and a control concept for analyzing manloading requirements during the maintenance phase.



Approved for public release, distribution unlimited

Dynamic Planning and Control of Software Maintenance: A Fiscal Approach

by

James F. Green
Lieutenant Commander, United States Navy
B.S., University of Utah, 1971

and

Brenda F. Selby
Lieutenant, United States Navy
B.S., Eastern Kentucky University, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1981

Authors: James F. Green

B. F. Selby

Approved by: Co-Advisor

Co-Advisor

Chairman, Department of Administrative Sciences

Dean of Information and Policy Sciences

ABSTRACT

Until recently, much of the budget planning for software systems has been primarily targeted at costs incurred during the development phase. However, with increasing software system life span and complexity, maintenance costs have become a more prevalent concern. As a result of necessary corrections for design errors and evolutionary maintenance, post-delivery investment in software systems now requires a greater proportional share of the life-cycle costs. In this research, various methodologies and system factors relating to software cost accounting are reviewed with the intent of developing a cost control model for arriving at a well-structured view for the management of the maintenance phase of the software life-cycle. The model proposed embodies a planning concept for establishing a maintenance strategy and a control concept for analyzing manloading requirements during the maintenance phase.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. THE PROBLEM

Recent literature is replete with dire predictions about the ultimate costs of software maintenance. In 1973, costs of software in the United States were $20 billion [1] and they are projected to be $200 billion in 1985 [2]. It has been hypothesized that anywhere from forty to ninety-five percent of the manpower effort in typical industrial applications occurs during the maintenance phase of the software life cycle. [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

Although there are numerous models in existence that deal with software costs, none deal specifically with the costs of 'pure' maintenance during the latter phase of the software life-cycle. It appears that much of the Federal Government and industry tend to use a general definition of software maintenance and treat it as a level of effort on various tasks rather than that effort allocated to specific tasks. Consequently, these organizations do not really know the true costs of their software maintenance.

The goal of this work is to investigate the methodology of software cost accounting, and to evaluate and develop a cost model for the prediction of pure software maintenance costs.

B. BACKGROUND

The term 'Software Maintenance' is very nebulous. Department of Defense Directive 5000.29 alludes to software maintenance by stating:

> "Correctness of software, reliability, integrity, maintainability, ease of modification, and transferability will be major considerations in the initial design." [15]

Used in this thesis is a composite definition of software maintenance to encompass those actions taken by a system user to retain an existing system in, or restore it to, an operable condition. This includes:

1) corrections to counteract detected bugs;

2) enhancements to add functions;

3) modifications to delete or change existing functions in their nature or scope;

4) implementation strategy to match changed conditions or requirements. [16, 17, 18, 19, 20, 21, 22, 23, 24]

'Pure' Maintenance on the other hand is restricted to that work accomplished during the maintenance phase of the software life cycle in the pursuit of the following goals:

1) Relialibility of Software - the ability of the software to produce consistent results whenever the customer uses the product;

2) Error Correction - changes made to counteract errors. The priority of correction is directly related to the seriousness of the error;

3) Software Maintainability - extending the useful life of a program by untangling a messy one, generalizing a specific one, or annotating an unreadable one.

Robert Glass [25] defines the best software maintenance as no maintenance at all. That is, no changes are needed because no errors were committed and all changes were anticipated. He then goes on to list six attributes of software maintenance:

1) Maintenance is intellectually very difficult. Problems cannot be bounded. The cause could be anywhere.

2) Maintenance is technically very difficult. Problems cannot be specialized. They could surface because of errors in the coding, design, architecture, or concept.

3) Maintenance is unfair. Usually the person who is maintaining a product did not write it and must interpret what the original author meant. Documentation is inadequate most of the time.

4) Maintenance is no-win. People only come to maintenance with problems.

5) Maintenance is infamous. There is very little glory, noticeable progress, or chance for 'success'.

6) Maintenance lives in the past. The general quality of code being maintained is often terrible. This is partly because it was created when everybody's understanding of software was more rudimentary, and partly because a great deal of code is produced by people before they become really good at programming.

Researchers do not appear to be using the same definition when working on the costs of maintenance. Those who estimate maintenance costs to be near forty percent of lifecycle costs seem to be using a definition closer to that of 'pure' maintenance while those who estimate maintenance as high as ninety-five percent are obviously using a very

general definition. According to recent surveys, most (seventy to eighty-eight percent) maintenance effort is spent modifying software to accomodate changes and to improve software performance rather than to correct errors which were not discovered during systems development. [26, 27, 28] These surveys have been substantiated by analyses done on three large scale systems:

1) Pacific Telephone - Service Order Retrieval and Distribution System;

2) Bell Telephone Laboratories - Automated Repair Service Bureau System;

3) VISA, Inc. - Base II World Wide Credit Card Transaction Interchange System. [29]

Although hardware costs have decreased by over two orders of magnitude and programmer productivity has increased by one order of magnitude in the last ten years, the total costs of systems are continuing to rise with the greatest portion of effort and cost spent after development completion [30]. There appear to be four primary reasons for this phenomenon:

1) Maintenance is people intensive;

2) The number of systems has increased substantially;

3) The mission of the software seems to be expanding;

4) Average system life has increased from three years in 1960 to five years in 1970 to eight years in 1980. [31, 32]

A recent DOD study reports that development costs for Air Force avionics software averaged $75 per instruction while maintenance costs lie in the range of $4,000 per instruction [33] This indicates that ninety-eight percent of the life-cycle costs of that system are spent on software maintenance. Another study concludes that fifty percent of the costs for Navy Airborne Antisubmarine Warfare Tactical Software is spent on maintenance software [34]. As one can see, there are many different meanings of the term 'software maintenance' and as many different assessments of its cost.

The software industry does not appear to be unified in its approach to decreasing the high cost of maintenance. McClure states:

> "A solution that focuses upon the production phases of the software life cycle does not address the major portion of the maintenance effort... We must directly address maintenance issues rather than hope that they will disappear by improving the development process." [35]

Most of the literature expounds the theory that, to be done properly, software maintenance should be a conscious goal from the beginning of the software development process. Maintenance is all too often left out of planning considerations and then treated as a helter-skelter, uncoordinated activity rather than a planned, methodical, controlled necessary business function [36]. Long term planning, just as

in other disciplines, includes the provision of appropriate tools. There are two major categories of tools for maintenance. Technical tools encompass such things as compilers, traps and traces, dumps, comparators, editors, reformatters, and preprocessors. Administrative tools include problem reporting vehicles, status reporting vehicles, and documentation systems. [37, 38, 39]

Even with the knowledge and use of these tools, productivity, which is typically measured in software lines of code (SLOC) is substantially lower for maintenance programmers than for development programmers. According to Daly, maintenance productivity can be as low as twenty percent of development productivity [40]. There appear to be three main reasons for this phenomenon:

1) There is a stigma attached to the job of software maintenance. Management rarely rewards good work in doing maintenance as generously as good work in doing development. Both coworkers and management personnel act as though they hold maintenance work in low esteem. To survive, every person must have self respect. If a job is not perceived as important, a person probably will not perform to the best of his abilities.

2) Usually, maintenance personnel are not intimately familiar with the code that they are assigned to maintain. Typically, a maintainer is assigned responsibility for 30,000 SLOC [41]. Because documentation is quite often poor, the maintainer must study the code itself and try to understand what the original developer created and why he implemented it in that manner. Usually he must study a great deal more code than the affected area to avoid inducing bugs in a seemingly unrelated area by the fix that is implemented.

3) The wrong grade of people are typically used to staff maintenance efforts [42, 43, 44, 45, 46, 47, 48, 49] Traditionally, maintenance efforts are being staffed by less experienced personnel than development projects.

However, maintenance personel should be senior people because software maintenance is a microcosm of the entire software development process. The maintainer does a systems analysis of a problem area leading to a requirements definition. Donning a designer's hat, the maintenance person then outlines the impact of the change on the product. Being a flexible individual, the maintainer now codes the design solution. After the results of these efforts have been tested and verified, the revised product is finally released to the world. The maintenance person also plays a liaison role with the customer by explaining anomalous outputs, negotiating changes that are needed as opposed to those that are desired, and interpreting the computer's unique constraints. As you can see, the person who maintains a complex system should be a highly talented and motivated individual. [50, 51, 52]

C. RESEARCH METHODOLOGY

1. Literature Search

Manual searches and automated system searches of the literature showed little had been published in this field. Although there is a lack of published material that deals directly with a fiscal approach to planning and control of software maintenance, the researchers found a great deal that was very useful as background information and which helped to develop the theory for the planning and control model.

2. Telephone Conversations

Efforts to uncover informal sources that deal specifically with the costs of 'pure maintenance' failed. The following organizations were contacted in the course of the research, with no significant results:

Army Computer Systems Command, Ft. Belvoir, Va.;

DOD Computer Institute, Washington, D.C.;

FEDSIM, Washington, D.C.;

Homestead Software Support Facility, Homestead, Fl.;

IBM Federal systems Division, Gathersburg, Md.;

Kapur Associates, Danville, Ca.;

National Security Agency, T303, Ft. Meade, Md.;

Naval Security Group Activity, Skaggs Island, Ca.; and

NARDAC San Francisco, Alameda, Ca.

Maintenance tracking data, dealing with Goddard Space Flight Center projects, was obtained from the Data and Analysis Center for Software, Griffis AFB, NY. Unfortunately, the late arrival of the data and format incompatibility precluded inclusion of this data.

Unpublished documents describing a matrix management method of functional area analysis developed by Mr. Kyle Rone, IBM Federal Systems Division, Houston, Tx. were obtained and significantly contributed to the formulation of the final model.

D. ORGANIZATION OF THE THESIS

In this introduction the problem has been stated, its importance discussed, and it has been placed in the context of the overall computer system development process. Chapter two covers various aspects of the problems encountered when

estimating/determining the cost of software maintenance. This specific background material is needed to understand the models that will be presented in chapters three and four. Chapter three thoroughly discusses existing models in two areas: those that work with Norden-Rayleigh curves, and those that encompass complexity metrics. Chapter four gives the authors' model which is based on both macro-estimating (total system) and micro-estimating (unit composition) techniques. Finally, chapter five summarizes the thesis and puts forth, conclusions and recommendations.

II. QUANTIFYING SOFTWARE MAINTENANCE

A. THE SOFTWARE PROBLEM

There are two main reasons that maintenance has become a predominant cost in software systems. First, the volume of completed systems which require maintenance dominates the systems under development as more and more long-lived large systems are completed and delivered. Second, software systems require a considerably greater proportional investment in error correction and evolutionary maintenance after delivery than other engineering products.

Numerous technological advances have not solved software problems. They have increased the demand for software, and opened up opportunities to use computers in new applications which place increasingly severe demands on software technology. Often the tendency is simply to ignore these problems. Because these problems are both technical and managerial in their scope, a "systems engineering" solution is needed.

Unlike hardware operation and support models, where the cost of spares, maintenance manhours, material, training, etc., can be estimated based on some physical characteristics of the system, software maintenance effort is strictly

a function of manhours to perform the necessary action. Thus far, maintenance costs for software seem to be primarily an estimate by an expert, someone familiar with the changes to be made to a program, rather than putting certain parameters into a cost estimating relation and calculating annual maintenance costs.

Software maintenance costs cannot be ascribed to one specific agent or event but instead to the combined action of many factors. By reviewing some of these, the complexity of the problem can be better understood. In this research, an attempt is made to isolate areas that can be estimated by formulas and then to establish the mathematical relationships. As such, the following topics will be discussed as they relate to the maintenance function:

1) Software Life-cycle;
2) Life-cycle Interrelationships;
3) Software Evolution;
4) Productivity;
5) Complexity Metrics; and
6) Error Prediction.

B. THE SOFTWARE LIFE-CYCLE

In the mid 1970's, the phrase "software life-cycle" was coined and became a popular means for conveying the basic concepts of a software system: multiple phases and extended life. Many representations of the life-cycle exist; by commonly accepted practice, the software life-cycle consists of the development phase and the maintenance phase taken collectively. Depicted in Figure 2.1 is a composite schematic showing this relationship.

This diagram oversimplifies the importance of the maintenance phase. A more accurate role of the maintenance function is detailed in the life cycle model (figure 2.2) developed by Rome Air Development Center [53]. From this view, maintenance performed during the operation and support phase is seen to be a highly interactive process. The conjecture apparent from the diagram is that the same procedural requirements for software development must be duplicated during the maintenance phase.

The basis of applying a life-cycle management scheme to software is to direct attention to all phases encompassed by the system life-cycle and the contribution of each phase to the total life-cycle expenditures. Familiarity with and

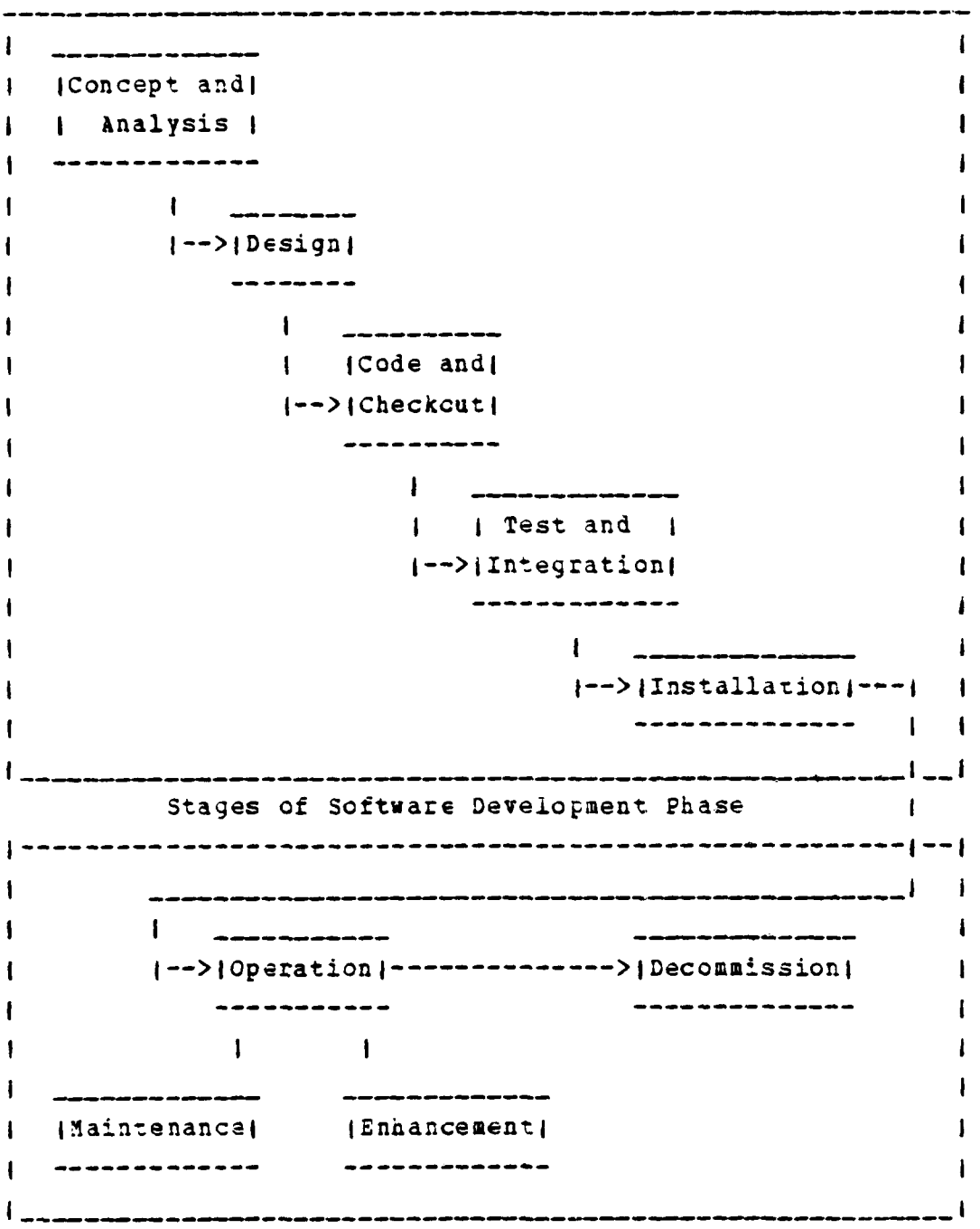


Figure 2.1. Software Life-cycle - Composite Schematic

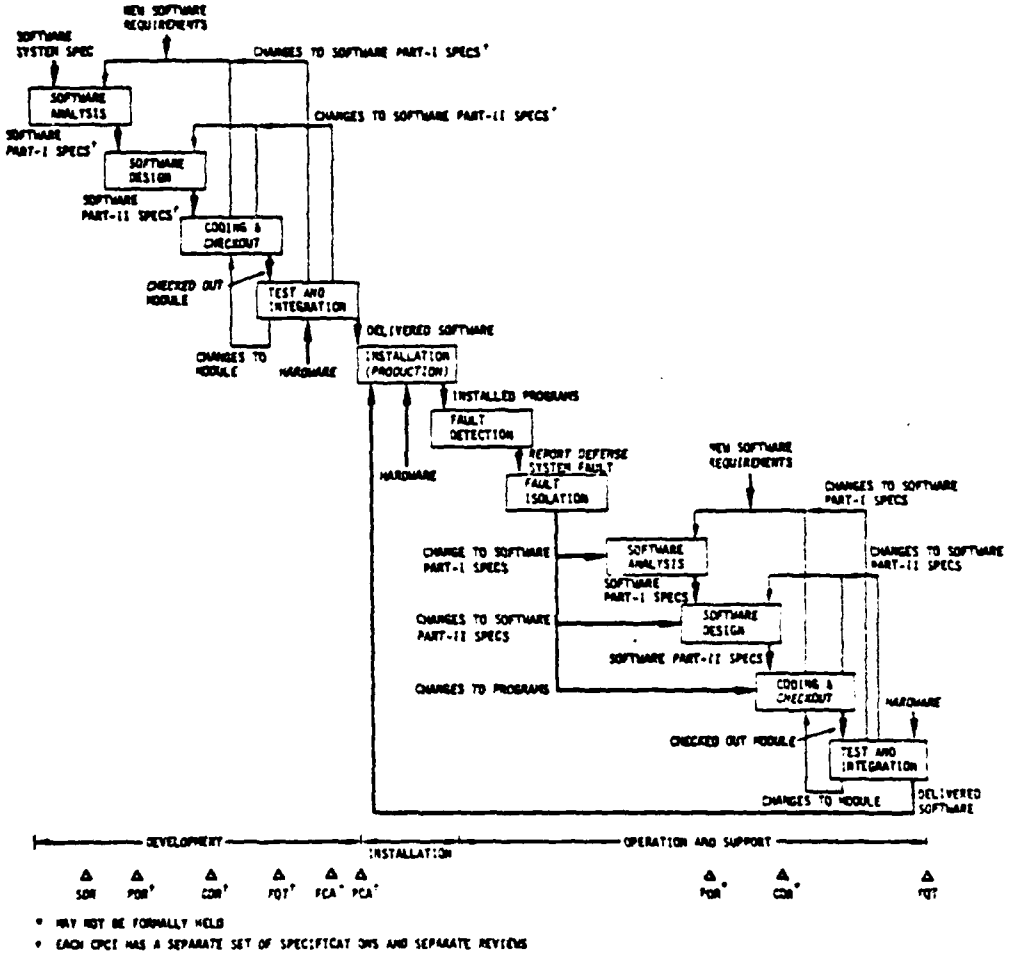


Figure 2.2. Software Life-cycle

understanding of the life-cycle can help managers make effective distribution of the resources for a software system which will ultimately effect the maintainability of the software.

The life-cycle curves, more recently called "Rayleigh" curves, were orginally formulated by Lord Rayleigh, the British Nobel Laureate. Presently, these curves are used to represent resource allocation (manpower) of a software project. Preliminary research in this area was directed at resource consumption in research and development (R&D) projects. In a series of studies conducted by Peter Norden [54] of IBM, it was established from a large body of empirical evidence that large R&D projects follow a life-cycle pattern as described by the Rayleigh (manpower) equation:

$$y' = 2Kate^{-at^2} \qquad (2.1)$$

where

y' = the number of person-years of effort expended per year,

K = the total number of person-years required over the life-cycle,

a = the curve shape parameter,

t = elapsed time in years, and

e = exponential function.

The principle of the curve is as follows: Research has indicated that there are regular patterns of manpower build-up and phase-out in complex projects. These patterns are made up of a small number of successive phases or cycles of work throughout the life of the project. Norden linked the cycles to obtain a project profile. When the individual cycles are added together, they produce the profile of the entire project (figure 2.3).

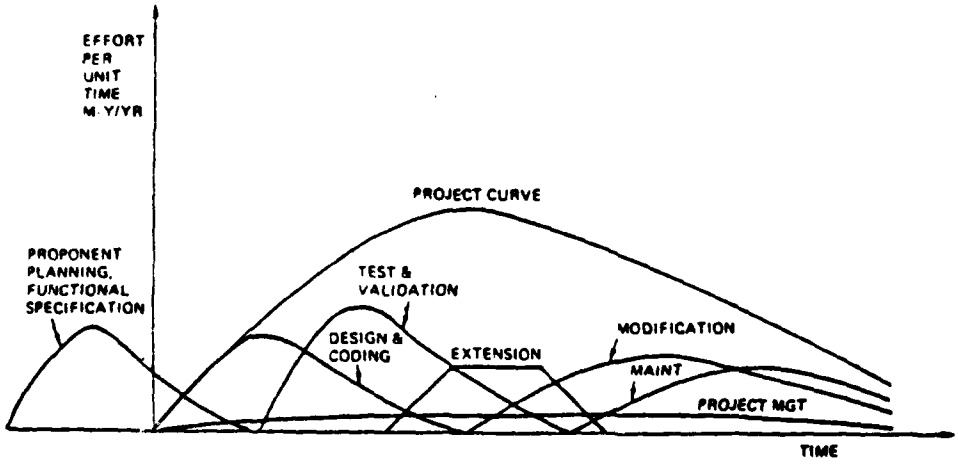


Figure 2.3. Project Profile

Peak manloading time (t_d) culminates during final stages of development and implementation (figure 2.4). Based upon

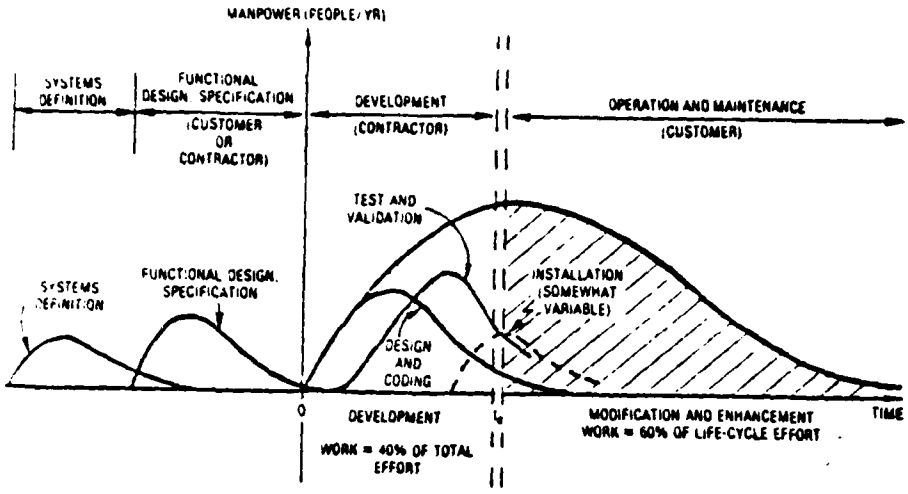


Figure 2.4. Manloading Profile

Norden's studies [55], cumulative resource allocation up to this time acccunts for approximately forty percent of the life-cycle. Occuring at the low end of the curve is the operation and maintenance phase which absorbs the remaining sixty percent of life-cycle expenditures. The greater portion of costs associated with this phase are attributed to the "maintenance tail" or expected life of the software product. Failure repair, however, is just a small part of post-delivery maintenance activities. Studies [5o] show that coding errors account for only thirty percent of the post-delivery errors. The greater share (seventy percent) is occasioned because there is a mistake in design or specification. Although the code performs exactly as designed, this does not reflect the original operational desires.

Logically, it would seem that maintenance manpower requirements would decrease over time due to growth in reliability. In other words, as programming and design errors, which are commonly called "bugs", are found and corrected, the time to the next system failure should increase throughout the maintenance phase of the life-cycle. This reliability assumption, however, is disputable. Maintenance action taken in response to error occurance can have three possible outcomes:

1) the actual error is corrected;

2) the error is corrected, but the fix induces a new error;

3) the error is not corrected, and the program remains non-operational.

Reliability growth, then, is a probabilistic event which depends heavily on the skills of the maintenance programmers. If the maintainers are competent, reliability should grow.

Another controversial assumption is growth in maintainability. When maintainability is viewed as the time required to return a software system to an operating status following a system failure and maintainability growth is viewed as the decrease in time required to correct an error, then an obvious conclusion would be growth in maintainability. Several factors, however, may produce an opposing

conclusion, i.e. decaying maintainability. Patchwork fixes, in addition to introducing new errors, may produce module interface problems and documentation inefficiencies will complicate the finding of other errors. Reduced familiarity with a software system, stemming from frequent personnel turnover, can be an inhibitor. Documentation and software (programming) standards and controls may not be enforced on new releases. Error identification and correction may become further entangled when configuration control is lax. Again, the competence of the maintainers will influence the results.

C. LIFE-CYCLE INTERRELATIONSHIPS

The management process for the maintenance of software involves decisions in establishing control of changes to the software and in providing for the implementation of improved functional capability throughout the life-cycle of the software. The planning to acquire and implement resources for software maintenance must:

1) consider the entire life of the software, and

2) begin early in the life of the software in order to reserve funding and identify sufficient resources for the future.

Different time spans and levels of effort exist for the different phases of a software project. The failure to obtain quantitative relationships of a precision comparable to those available for estimating the costs of hardware systems has led to the belief that interrelationships exist among life-cycle phases. That is, the amount of resources used in early phases impacts heavily on the resource requirements for later phases. Using an approach similiar to basic economic production theory, Thibodeau and Dodson [57] developed a mathematical model to describe the complexity of the phase interrelations. This relationship is given in the form:

$$Q = AK^{a}L^{b} \tag{2.2}$$

where

Q = the level of output,

K = the amount of capital input,

L = the amount of labor, and

A, a, and b are empirically derived constants.

Graphically, this is shown in figure 2.5.

To add a term representing technological change or to account for different classes of labor or capital, the number of input resources can be expanded to

$$Q = AK_1^{a1}K_2^{a2}L^{b} \tag{2.3}$$

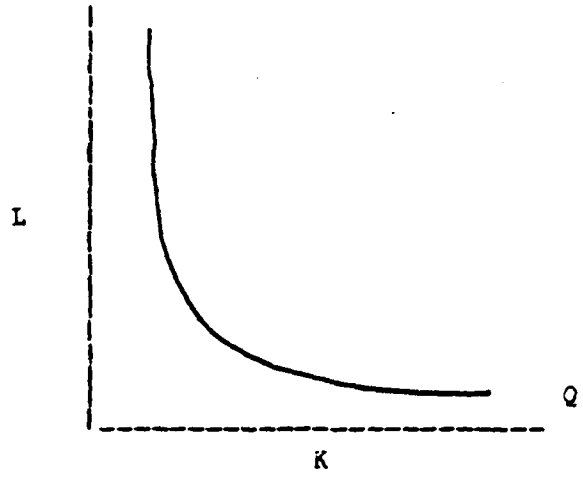


Figure 2.5. Economic Production Curve

In order to indicate trade-offs between life-cycle phases, the same general formulation can be used and expressed as

$$P = aX_d^b X_c^c X_t^d X_m^k \qquad (2.4)$$

where

P = software production resulting from the application of the resources,

X = person-months of inputs,

a, b, c, d, and k are empirically derived constants, and

subscripts d, c, t, m represent designing, coding, testing, and maintenance respectively.

A further assertion made by Thibodeau and Dodman indicated that limitations in design resources (e.g. a reduction

in planned resources) may be passed through the development phases with final impact in the maintenance phase (higher error rates). Based on the mathematical postulate previously described, this type of relationship can be shown by the graph in figure 2.6.

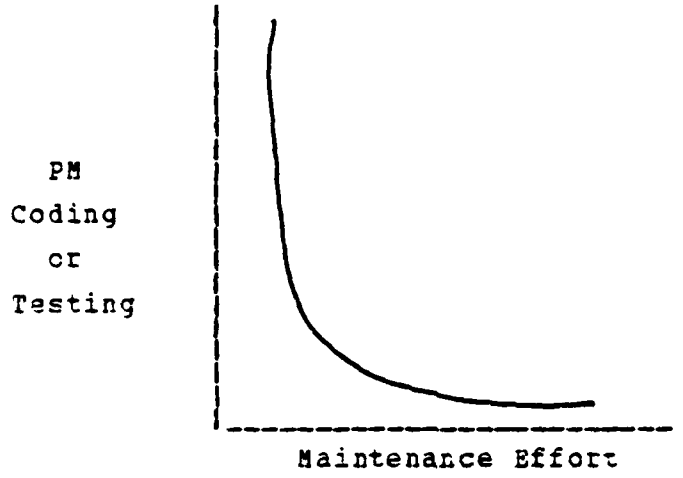


Figure 2.6. Application of Production Theory

In describing the infinite set of relationships, table I illustrates some departures from the ideal which may occur, and how a reduction or increase of resources in these phases will be reflected in the error rate of the delivered software. While it can be argued that the ideal error rate may be zero, a more practical solution would be to avoid dedicating an enormous amount of resources to achieve zero errors. As a result, it would be expected that for most

information systems, planning would allow for some marginal error rate. However, this tolerance of errors does not necessarily apply to tactical defense systems.

D. SOFTWARE EVOLUTION

Operational software systems undergo a continuing process of evolution, the phases of which are repair, modification, enchancement, and adaptation. Continuing evolution is the visible sign of continuing interaction between the system and its environment. Even if --and this rarely, if ever, occurs-- its first implementation was perfectly conceived, perfectly designed, and perfectly implemented, a program will require general maintenance.

Evolution dynamics is a theory describing the change of a software system over a period of time. The theory distinguishes between <u>progressive</u> work (to introduce new features) and <u>antigressive</u> work (fault correction, testing activity, and investment in methodology to combat the complexity which grows with system size) [58]. The basic assumption of programming evolution dynamics is that it is legitimate and necessary to view a large program and its maintenance organization as interacting systems. Thus one must search "for models that represent laws that govern the dynamic behavior

TABLE I

Hypothetical Phase Interrelationship Trade-offs

	Hypothetical cases							
	1	2	3	4	5	6	7	8
Analysis and design	>	=	=	<	=	>	=	=
Coding and checkcut	=	<	<	=	=	>	<	<
Testing	=	=	>	=	=	>	<	=
Maintenance	=	=	=	=	=	=	>	>
Changes	No	No	No	No	Yes	Yes	No	No
Reported error rate	<	>	=	>	>	=	>	>

Symbols:

= equal to ideal

> greater than ideal

< less than ideal

of the metasystem of organization, people, and program material involved in the creation and maintenance process" [59, 60].

Feedback is basic to the process since the system and system designers are considered as a metasystem. The key to good feedback is intensive use over time. The more the software is used, the better it gets, as long as deficiencies are fed back into the maintenance group and corrections are made. This statement holds true provided that the maintainers introduce fewer errors than they resolve. Likewise, the longer it is used the less the probability that the system contains major deficiencies. In analyzing a software development system, a simple beginning would be as shown in figure 2.7. When pressure is exerted to provide bigger releases (later versions of a system that contain enhancements and/or corrections), the results are more complexity, reduced quality, and growth rate limiting factors. Eventually, releases are made solely for restructuring/rewriting. At this point, a fission effect is possible where excessive growth leads to system breakup.

Various published papers [61, 62] have discussed the characteristics and dynamics of the evolution of large

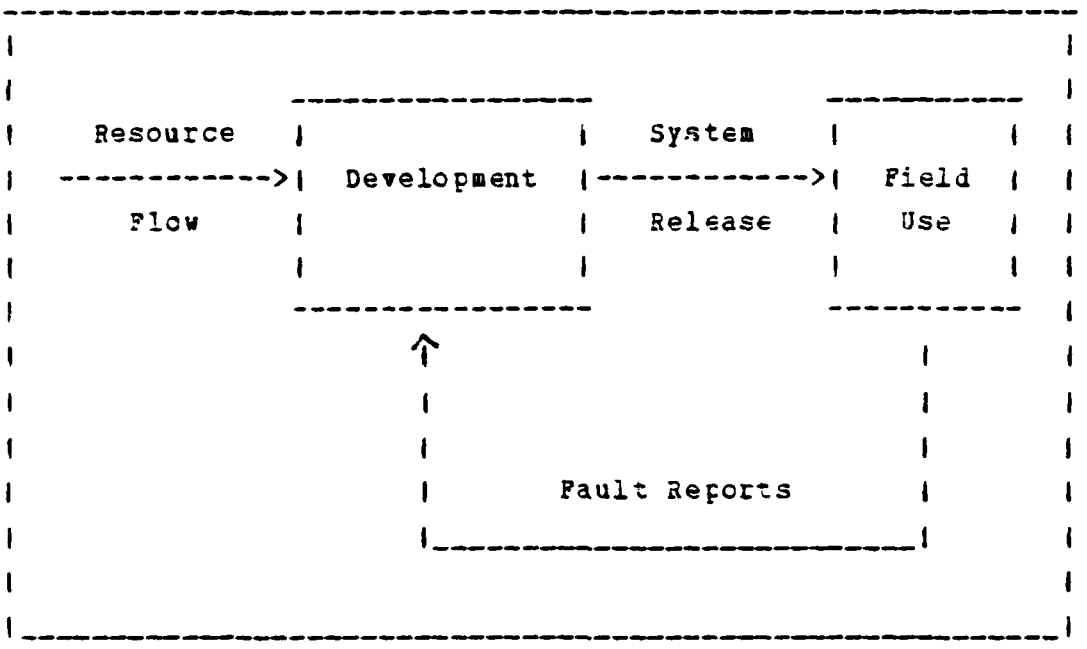


Figure 2.7. Development Release Cycle

programs, with the most significant contribution of research done by Lehman and Belady [63, 64]. Their efforts were directed at understanding the dynamics of the software lifecycle, thereby creating an enhanced environment of managerial awareness and an understanding of system behavior. Long term unpredictability of the system development and maintenance processes have been attributed to the human interface. However, it has been found that measures of system activity such as number of modules handled, inter-release time, and total number of modules in the system, show an unusual

regularity. Since this regularity could not be attributed to management decisions, Lehman and Belady have tried to analyze it through the use of evolution dynamics. By describing the environment of program creation and maintenance in terms of regularities, trends, and patterns, they have proposed laws governing the evolution dynamics (table II).

Features of these evolutionary trends were further supported in a more recent study by Leintz and Swanson [65]. Analysis of data obtained from an extensive survey indicated that the magnitude of a maintenance effort can be explained by the combined efforts of four variables: system age, system size, relative amount of routine debugging, and the relative experience of the maintainers. The relationships of these variables were modeled as shown in figure 2.8. Amount of maintenance effort, the dependent variable, is seen to be influenced through five other causal paths involving four variables. Each causal path is initiated from the independent variable, system age.

E. PRODUCTIVITY

Productivity is often considered a measure of the transformation of meaningful and controllable units of input to meaningful and controllable units of output. The question of

TABLE II

Laws of Evolution Dynamics

CONTINUING CHANGE A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.
INCREASING COMPLEXITY As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.
THE FUNDAMENTAL LAW **(OF PROGRAM EVOLUTION)** Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends/invariances.
CONSERVATION OF ORGANIZATION STABILITY **(INVARIANT WORK RATE)** The global activity rate in a project supporting an evolving program is statistically invariant.
CONSERVATION OF FAMILIARITY **(PERCEIVED COMPLEXITY)** The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

quality must be understood in all measures of productivity, if they are to have meaning. It is far easier to be more productive when producing throwaway products than it is when producing high quality output.

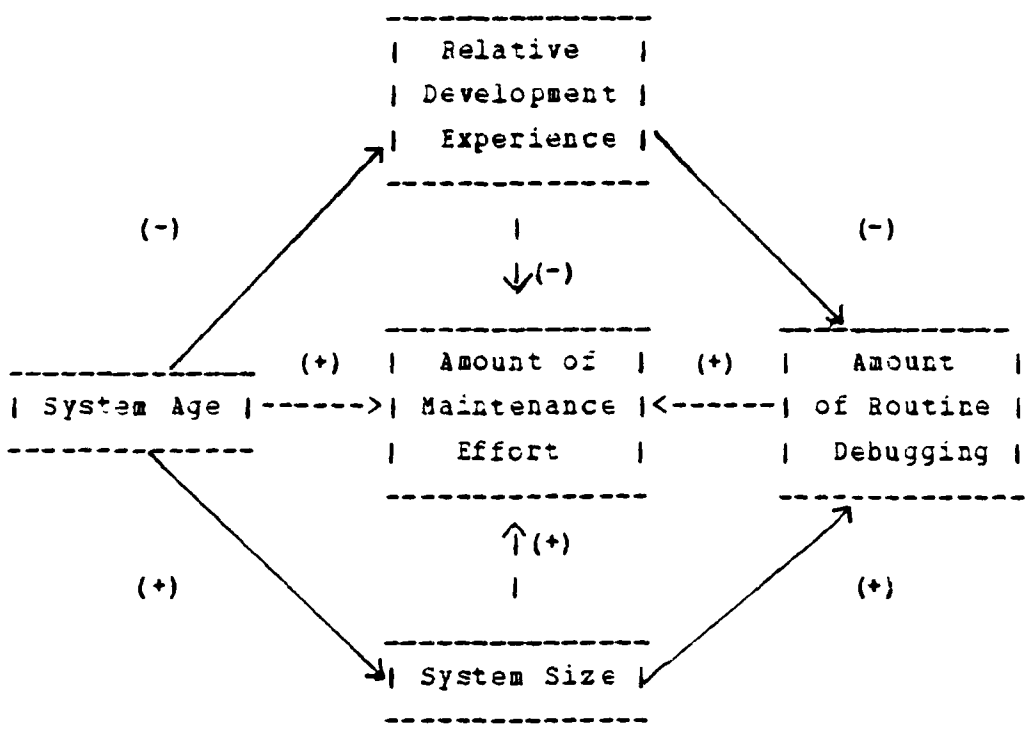


Figure 2.8. Casual Paths of Maintenance Effort

If software is sized in terms of a product measure such as the number of instructions or modules, then the assumed personnel productivity against those measures is a key variant in the estimate. Since producing software is a very labor intensive activity, consuming greater than eighty five percent of the resources allocated for software development [66], an essential ingredient for arriving at an accurate cost estimate of the software lies in personnel productivity. Generation of software is creative and, therefore, a wide variance across personnel productivity can be expected.

Budget estimations required for software development have led to an abundance of research exploring the topic of programming productivity [67, 68, 69] Traditional measures of software productivity have included:

1) dollars per defect,
2) lines of code (LOC) per person-month (PM),
3) dollars per LOC,
4) dollars per PM, and
5) complexity branch per 1000 LOC.

Maintenance researchers pose the yet unanswered question: Can the same criteria be applied for productivity during the maintenance phase? Within a maintenance scenario, module constituents of a software application can be categorized as new, modified, retained, and converted (figure 2.9). New segments consist of entirely new code. Modified segments are composed of changed code and the unchanged code that may be affected by the changed code. Retained code consists of previously developed and tested segments that will be integrated into the software products without being modified. Converted code is existing code converted to another language. Each of the categories of code, when related to a specific product, may produce a unique productivity rate.

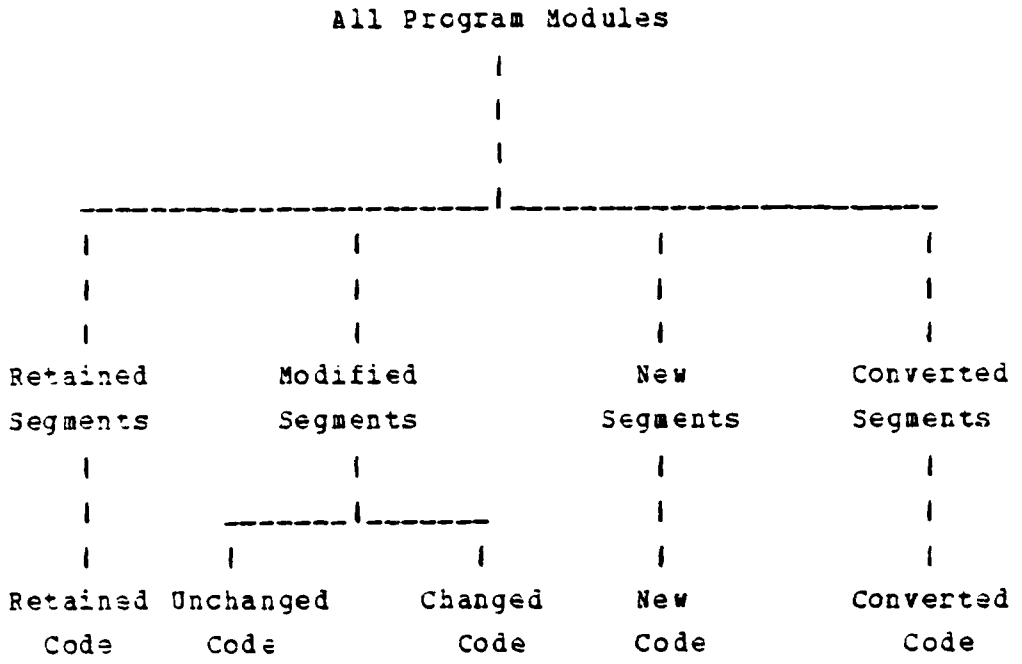


Figure 2.9. Categories of Program code

Factors which influence productivity have been widely researched. Data collected from sixty projects by Walston and Felix showed that significant relationships existed between productivity (SLOC) and the ratio of develcped code to the sum of original (or reused) code plus the developed code [70]. The resulting plot shown in figure 2.10 suggests that productivity is highest when there is no original or reused code, that is, when all the code is developed from the inception of the project. As the percentage of reused code grows, the expected productivity decreases.

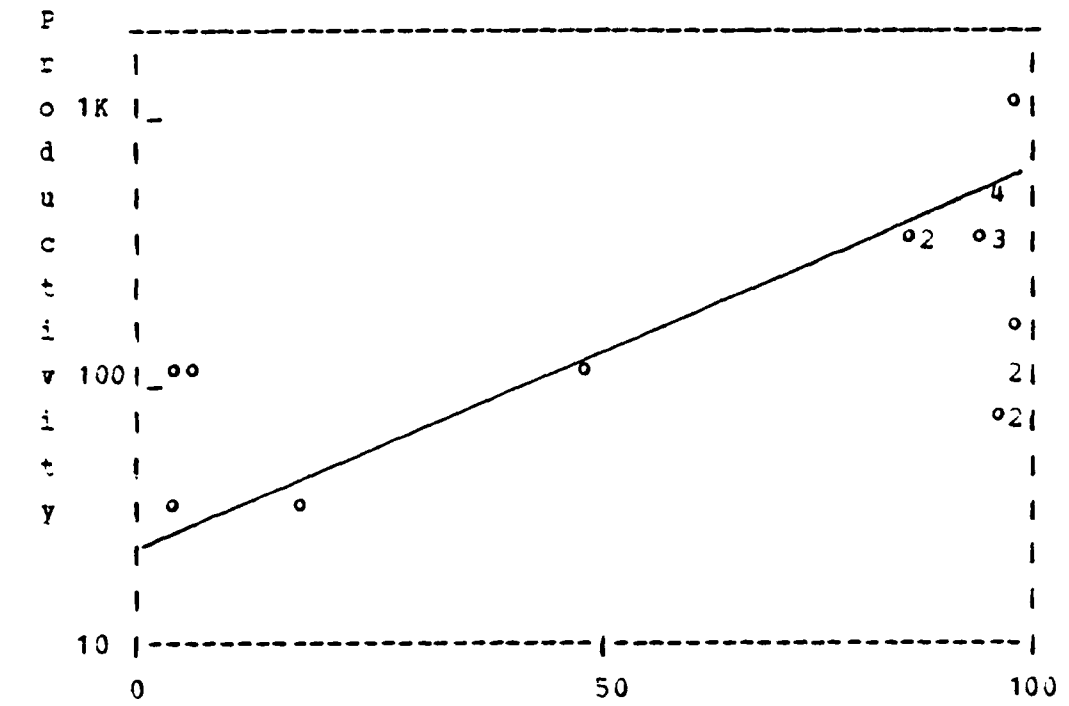


Figure 2.10. Productivity - Reused Code Relationship

Recent investigations done by Swanson and Leintz, revealed that while productivity techniques have been extensively discussed, few systemic studies of benefits in the maintenance phase have been conducted [71]. Figure 2.11 shows some, but not all, of the factors commonly cited as indices of productivity.

Maintenance costs must be viewed collectively with productivity. To do less is to focus on only part of the

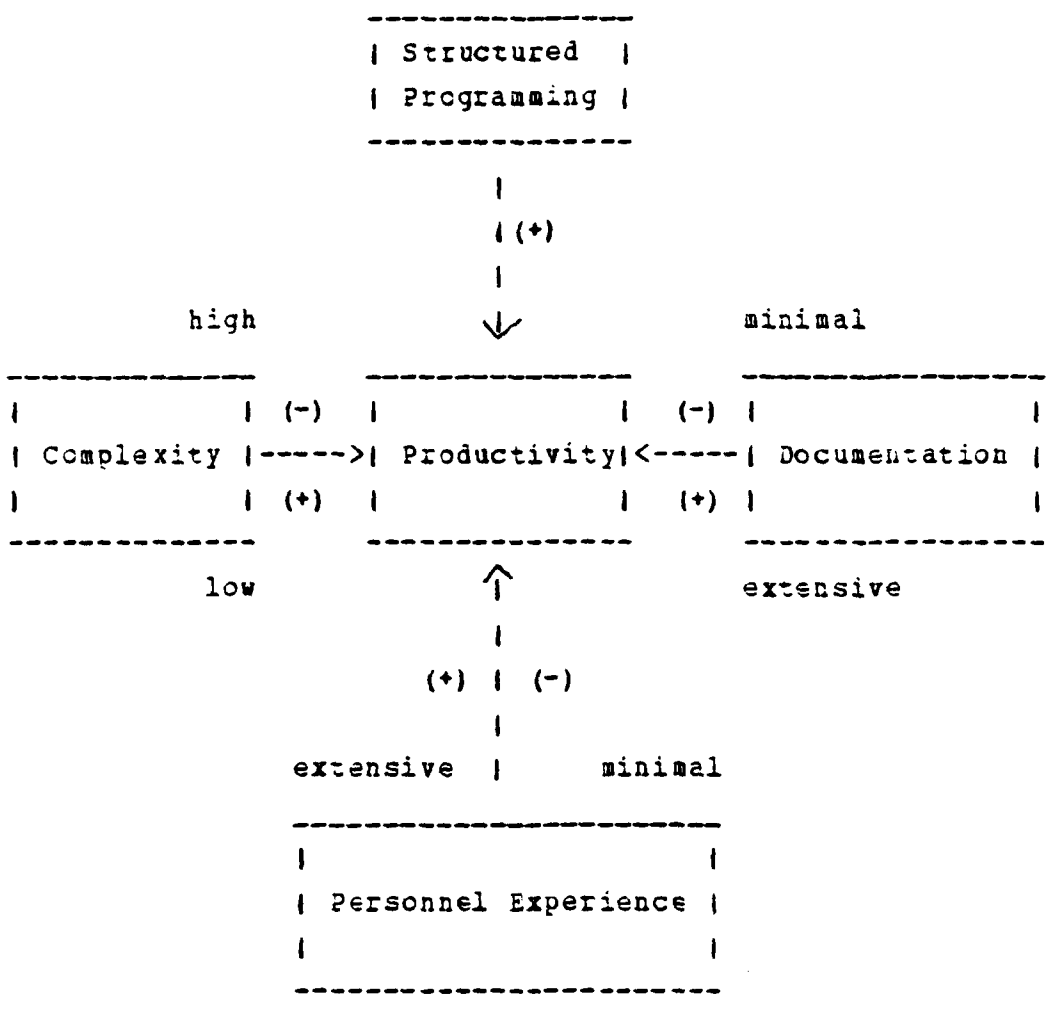


Figure 2.11. Productivity Determinants

issue. It could be a misleading focus if management dictates policies that result in high productivity during development work but adversely affect the productivity of post-delivery maintenance. If the productivity is negatively affected because of internal problems prior to delivery

or reduced quality when in use, then costs will increase and affect the potential to complete other projects.

F. COMPLEXITY METRICS

Quantitative metrics which assess the complexity of software continue to attract a high degree of interest. These metrics are assumed to be valuable aids in determining the quality of software. A collection of such metrics which assess numerous factors that constitute this nebulous "software quality" have been proposed [72, 73]. Such factors include reliability, portability, maintainability, and myriad other xxx-abilities.

Potential uses for measures which assess these various factors are manifold. Importance of metric relationships lies in the following areas:

1) As feedback to programmers, they can be used during development to indicate potential problems with developed code [74]. A design is evaluated with the metric relationships in mind. If it appears that this design falls outside of the metric bounds, then another design must be contemplated.

2) In guiding software testing, McCabe's cyclomatic number has been proposed as a means of assessing the computational complexity of the software testing problem [75]. Other metrics which index the quality or complexity of software may help identify modules or subroutines which are likely to be most error-infested.

3) If one of a combination of metrics can be empirically related to the difficulty programmers experience, then more accurate estimation can be made of the manpower that will be necessary during maintenance.

In using these metrics, it is important to distinguish between the computational and psychological complexity of software, since reasons for assessing them differ. Computational complexity refers to "the quantitative aspects of the solutions to computational problems" [76] such as comparing the efficiency of alternate algorithmic solutions. To illustrate, as the number of distinct control paths through a program increases, the computational complexity may increase. Psychological complexity refers to characteristics of software which make it difficult to understand and work with. Psychological complexity can then be thought of as assessing human performance on programming tasks. Subsequent sections will discuss currently used metrics that have been coupled with the maintenance effort in an attempt to predict programmer effort required to complete a specific maintenance task.

1. Halstead's E

During the last few years research aimed at the development of a "software science" has supported the contention that there may be simple theoretical explanation for the structural characteristics of many computer programs and that there is a strong quantitative relationship between

these characteristics and the effort required to write programs [77, 78, 79]. Based on the theory of software science, five entities of an algorithm expressed in a language are measureable:

n_1 = number of distinct operators,

n_2 = number of distinct operands,

N_1 = total number of operators,

N_2 = total number of operands, and

n_2^* = number of input/output parameters for the algorithm.

From these measurements, some defined properties for programs can be calculated: length (N), vocabulary (n), volume (V), and program level (L). [80]

Using the simple relationships between these metrics and the effort (E) required by a programmer, Halstead arrived at an expression of effort (total number of elementary mental discriminations) to generate a given program where

$$E = \frac{V}{L} = \frac{n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2)}{2n_2} \qquad (2.5)$$

By applying the Stroud number, which is the number of elementry pieces of data that a person can mentally seperate per second (S), a dimension of time is introduced to the effort equation:

$$T = \frac{E}{S} = \frac{V}{SL} \qquad (2.6)$$

where T indicates the estimated time for programming. Except for the Stroud number, all parameters on the right side of the equation are directly measureable for any implementation of an algorithm. Research methods using calculated T values have shown that a strong correlation exists with the actual time measurements in the absence of certain "impurities" which correspond to common undesirable programming practices such as unstructured code, low module cohesiveness, high module coupling, etc. [81].

2. McCabe's v(G)

More recently, T. McCabe [82] developed a complexity definition based on the decision structure of a program. McCabe's metric is the classical graph theory cyclomatic number v(G) defined as:

v(G) = number of edges - number of nodes
+ 2(number of connected components).

Two simpler methods of calculating v(G) are presented by McCabe: the number of predicate nodes plus 1 or the number of regions computed from a planar graph of the control flow.

Literally, this complexity metric counts control path segments which, when combined, will generate every

possible path thrcugh the program. Since additional control paths could make a program more difficult to understand, the number of basic paths indexed by this metric may also relate to mental difficulty of a programming task.

G. ERROR PREDICTION

If managers knew how a program behaved for every conceivable ccmbination cf inputs and could accurately predict all future input combinations, then they would knew precisely how many errors are in that program and could predict at which point in time that the program would next fail. As a result, it would be fairly simple to program resources for software maintenance. The only real decision, then, would be whether the annoyance from the error was worth the effort to eliminate it. Because this ideal situation is not a realistic representatien of the world, except in the most trivial programs, it would be a great aid to managers to have a method to predict residual errors with a reasonable degree of certainty. This capability would arm them with a good guide for programming the amount of maintenance effort needed for the next time period.

In the early days of computing, managers obtained rough estimates of the number of errors in a module by assuming

that there was one bug in every sixty lines of code or perhaps in every one hundred lines of code depending on each manager's optimism and experience [83]. It seems to be a reasonable assumption that there is a better way to predict residual errors. The importance of error detection analysis has been recognized in the past few years and many studies have addressed this problem. [84, 85, 86, 87, 88, 89, 90, 91] An important objective of most of this work has been to develop analytical techniques to examine the error phenomenon in order to compute or predict items of interest such as the number of errors detected at time t, the presumed number of remaining errors at time t, and the software reliability function. (It should be noted that none of these studies deals specifically with the detection or the prediction of errors during the maintenance phase of the software life-cycle.)

One would expect software reliability to improve with age because latent bugs are detected and are presumably corrected. However, there are exceptions to this general statement. Bugs can be induced into programs while corrections are being made. This situation, called the "ripple effect", generally happens in very large systems like O/S360 instead

of small systems like a compiler [92]. When a change is made in module 'A' it affects the way module 'B' works. The maintainer has neither the desire to change another module nor, probably, any idea that his change would affect another module. With vast, complex systems it is impossible for any person to know all of the ramifications of a change. Since most operational software is subject to enhancements and changes in requirements because of the dynamic environment in which it is run, it can be expected that bugs will be induced with the new code and that other modules will be affected through interfaces with the new modules. In the long run however, it appears that most software projects follow the predicted process and have fewer errors as time elapses [93]. Table III [94] provides data to support this phenomenon. Observe the great variability of the data and the increased reliability as time passes.

Although the code appears to become more reliable as time passes, there are still problems with error prediction models. Many of these models assume a constant error rate [95, 96, 97, 98, 99]. This does not strike one as being a reasonable assumption on three accounts. First, the failure rate will fluctuate because the frequency of execution of

TABLE III

Successive Execution Times Between Failures

(Measured in seconds, read left to right and top to bottom.)

9	2	91	112	15
138	50	77	24	108
88	670	120	26	114
325	55	242	68	422
180	10	1146	600	15
36	4	0	8	227
65	176	58	457	300
97	263	452	255	197
193	6	79	816	1351
148	21	233	134	357
193	236	31	369	748
0	232	330	365	1222
543	10	16	529	379
44	129	810	290	300
529	281	160	828	1011
445	296	1755	1064	1783
860	983	707	33	868
724	2323	2930	1461	843
12	261	1800	865	1435
30	143	109	0	3110
1247	943	700	875	245
729	1987	447	368	446
122	990	948	1082	22
75	482	5509	100	10
1071	371	790	6150	3321
1045	648	5485	1160	1864

the areas of code varies. Some areas may never be executed [100]. As an example, if one assumes that there are one hundred bugs in a program, that the failure rate is fifty failures a week, and that one is using a constant error rate prediction, then after fifty bugs have been eliminated the failure rate should be to be twenty-five failures per week. If the bugs are eliminated in the order that they are detected, the first fifty to be eliminated would be in the most frequently exercised areas of code and the observed failure rate would be less than twenty-five per week. If, on the other hand, the most severe errors were corrected first, there may be a situation where there are several annoying but non-critical bugs in a highly exercised portion of code and the observed failure rate is forty failures per week despite having eliminated fifty bugs.

Second, according to Ottenstein [101], the error rate for modules, at the validation and integration stage, varies inversly with the size of a module. This theory has been corroberated by Motley and Brooks [102]. Motley and Brooks feel that this inverse proportion is an indication that the larger modules were not as fully debugged during the

validation and integration stages and would go into the operations and maintenance phases with a greater proportional amount of errors. Ottenstein explained the phenomenon in just the opposite manner. She feels that there is a learning and retention benefit that operates with large modules and thus the larger modules will go into operations and maintenance with a smaller proportional amount of errors.

A third reason for a variable rate of errors at the validation and integration phase is also proposed by Ottenstein [103]. Earlier developed modules are more fully debugged in the initial testing because at that period in the project there is a lot of time and money to do the job correctly. However, modules that are developed near the end of a contract appear to be hastily and incompletely debugged before being submitted for validation because both time and money are running out. The authors propose a corollary to this hypothesis. The more over-budget and behind-schedule that a project is delivered, the higher should be the prediction of errors detected in the maintenance phase.

Even if a manager could accurately predict the number of errors that will be detected in a given time period, there would still be a problem in scheduling the proper amount of

resources. Different types of errors will require different amounts of effort for correction because they are of different complexities.

H. CHAPTER SUMMARY

Numerous software topics are under study in an attempt to uncover explanations for the phenomenology of the software life-cycle. Of more specific concern are the events which lead to the increased expenditures during the operation and maintenance phase of the software project. Indications from research evidence are that not one single factor can be named as the dominant contributor to the life-cycle maintenance costs. Instead, a multiplicity of factors are cited as having an impact on the total system.

Recognizing the futility of identifying a single contributor, reseachers have resorted to finding the control elements that best define the changes that occur in system characteristics. With a continued research effort, better understanding and increased familiarity of these system control elements may lead to positive results in linking system characteristics with maintenance requirements.

III. COST ESTIMATION OF SOFTWARE MAINTENANCE

Coupling the rising costs of computer software with the relative decline in computer hardware costs would indicate that computer software acquisition cost and maintenance and operation cost (collectively referred to as software life-cycle costs) constitute the greatest share of the data processing budget. Consequently, predicting future software costs for both existing systems (maintenance and operation costs) and new development is of increasing concern to management.

The phenomenology of the software development and maintenance process is not definitively known. Research suggests the existence of a fairly clear time-varying pattern such as a Rayleigh curve or some other similiar form. The analysis is complicated by the presence of "noise" or stochastic components. Additionally, the observable components (manpower, cost, time) are strongly subjected to management perturbation. This would indicate that although a system has a characteristic life-cycle behavior, if that behavior is not known to managers *a priori*, then they will respond reactively (non-optimally with time lags) to system demands. A reasonable basis now exists for expecting that

an adequate phenomenological description may arise from the following sources:

1) statistical mechanics;

2) information theory coupled with statistical communication theory;

3) diffusion and transport theory. [104]

Tracking of costs throughout the life-cycle is important because, as pointed out in chapter Two, sixty percent of the life-cycle effort is consumed during the operations and maintenance phase. If this phase is treated as a level-of-effort task, then far more resources than necessary for maintenance are used. Given a fixed manpower or budget constraint (very common in government), less than optimal control of the work during this phase increases the possibility of maintenance work saturation (i.e. devoting all resources to maintenance). This situation leaves no capability to accomplish additional work.

Within the scope of this discussion, three types of models for addressing maintenance cost estimation will be considered:

1) software cost estimation from a macroestimating view using the Norden-Rayleigh curve parameters;

2) software cost estimation from a microestimating view using a work breakdown structure(WBS) methodology;

3) software evolution dynamics using system complexity as a cost monitor.

The format of presentation will include a general description of the model with subsequent application of the model to the forecasting of costs within a maintenance scenario.

A. SOFTWARE COST ESTIMATING MODELS

1. Putnam's Software Cost Estimating Model

a. Description

This model attempts to provide quantitative answers to the questions often asked by managers about software projects. These questions are generally concerned with project time duration, total cost, and the accuracy of the figures presented. Putnam's [105] methods provide estimates in the following areas:

1) Total life cycle effort in manyears;

2) Cost for the project;

3) Peak manpower needed;

4) Manpower needed at any specific time or phase in the project;

5) Risk and variance analysis of derived estimates; and

6) Linear programming (LP) techniques to impose real world management constraints.

Putnam's contribution to software cost estimating was to apply the Rayleigh curve to software life-cycle manloading. Using the techniques based upon the life-cycle

theory developed by Norden, Putnam did a number of empirical studies and found that the software life-cycle exhibits a rise in manpower up to a peak and then a trailing off.

Basically, the Putnam model obtains estimates of the measure of work in man-years and of the total development time of the project. Development time in the Putnam model is defined as the elapsed time on the project up to the point when the system reaches full operational capability, but not including the system definition and functional design/specification phases. The estimates of the total life cycle in man-years and the development time are then used to derive an equation giving the ordinates for a man-power expenditure curve for a specific project. A yearly dollar costing can then be computed for the project by multipying the ordinates of the man-power curve at each year by the average cost/man-year to arrive at a dollar cost/year and, subsequently, at a total dollar cost for the project. Putnam uses the Raleigh equation, which has been empirically determined to fit the project manpower loading profile for large projects and to best represent Norden's model. The most popular form of the curve is the derivative form and is expressed by:

$$y' = 2Kate^{-at^2} \tag{3.1}$$

where

y' = the number of man-years of effort expended per year,

K = the total number of man-years required during the life cycle of the project,

a = the curve shape parameter,

t = the elapsed time in years, and

e = the exponential function.

With the assumption that the shape of the curve is somehow related to both the difficulty of a particular development and to the skill level of the project team, a means for expressing these relationships in terms of Rayleigh curve parameters was derived. The relationship of the parameter a to development time (t_d) is:

$$a = 1/2t_d^2 \tag{3.2}$$

which, when substituted into the derivative form of the Rayleigh curve, results in the following equation:

$$y' = K/t_d^2 \; te^{-(t^2/2t_d^2)} \tag{3.3}$$

To use the above equation, estimates must be found for the total life cycle in man-years (K), and the development time (t_d). Virtually every parametric software cost model is based on an estimate of computer program size, measured in either source statements or object code instructions. Putnam uses source statements because that is what programmers produce. Likewise, it simplifies the mathematical computations because compiler efficiencies are not considered. The relationship that is used by Putnam to equate source statements to development time and project effort is given by the following equation:

$$Ss = Ck * K^{(1/3)} t_d \qquad (3.4)$$

where

Ss = delivered source lines of code, and

Ck = state-of-technology constant.

Within the model, estimating program size is viewed as an iterative process that should be recomputed several times during the system definition and functional design/specification phases in the software life cycle. The first estimate is done at project conception and can be little more than a best guess used to establish basic economic

feasibility based on past software projects and expert opinion. As more knowledge is gained about the project, individual segments of the system are estimated seperately and then totaled to give a more accurate estimate of the expected size. Also, standard deviations and confidence intervals are derived from statistical methods that use best and worst case estimates.

To determine the technology constant, data from past software projects must be inserted into the software equation (3.4) to derive the unknown variable Ck. It should be noted that Ck is initially very difficult to determine but should remain consistent for similar projects within a specific organization. After the parameters Ss and Ck are determined, various values of t_d and/or K may be substituted into the software equation to produce a parametric graph showing size versus effort and time (figure 3.1).

A constraint line determined by management and representing a difficulty gradient for certain types of projects is then superimposed on the graph. Values that fall below this line are determined to be infeasible for software development.

After values and ranges are found for t_d and k,

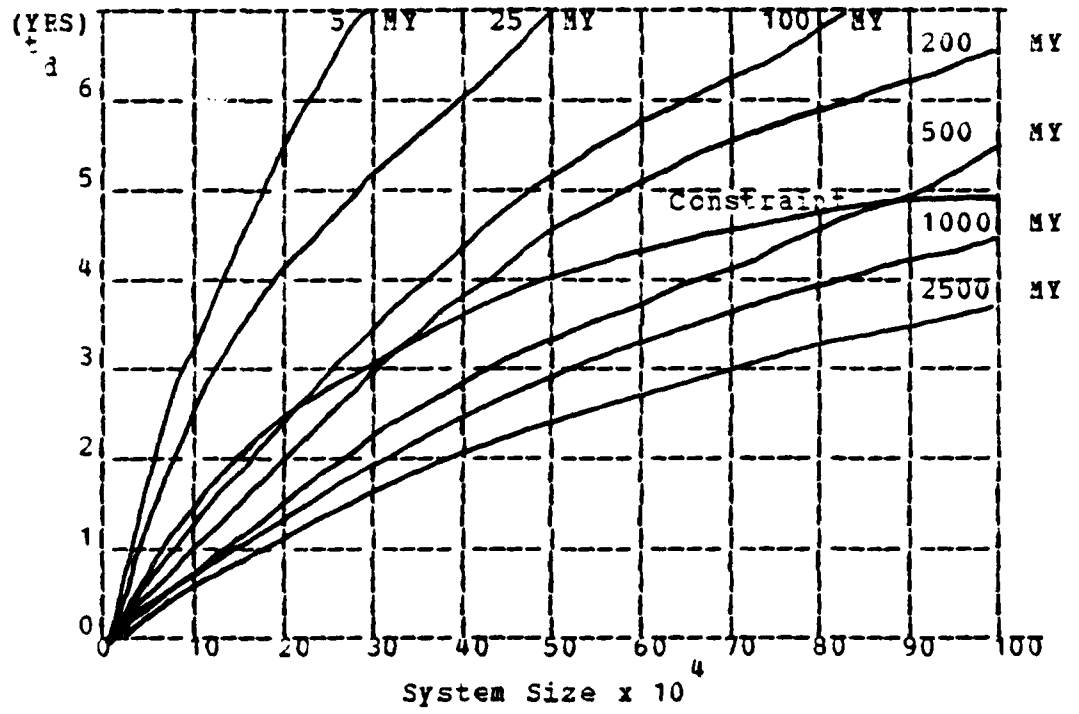


Figure 3.1. Size vs. Effort and Time Relationship

dollar costs for the project may be computed by multiplying plying an average labor rate per man-year by an expected value of man-years tc derive an estimated total cost for a project. A variance estimate for dollar costs may be obtained in a similar manner from the variance of man-years. While this model recognizes that real world managerial constraints exist, they are not explicitly addressed. Instead it is recommended that linear programming techniques should be used to account for everyday concerns such as

contract deadlines, cost ceilings, and hiring practices and capabilities.

b. Application to Maintenance Costing

Putnam's model takes a macro approach to answering the questions most often asked by managers concerning the areas of time, effort, and cost. According to relationships determined empirically, an overall estimate of man power is obtained and subsequently allocated among the different phases. To determine the risk involved in the estimation, statistical methods are used which give the manager a 'feel' for the accuracy of the data presented to him.

As work proceeds during the life-cycle, uncertainty about the management parameters decrease. In order to follow and track the time-varying behavior of a software system, empirical data must be collected and plotted to show the current labor force for any given time (figure 3.2). Using this data stream, time series analysis can be done. By turning the characteristic Rayleigh behavior into a straight line, the actual manpower data may be fitted to get a revised estimate of future resource consumption.

The linear form of the Rayleigh-Norden curve is illustrated in figure 3.3. This form may be obtained by dividing equation 3.3 by t and taking the natural logorithm of

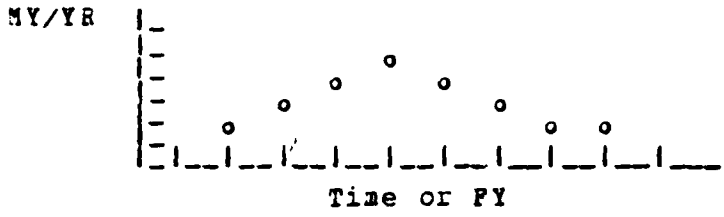


Figure 3.2. Typical Plotting Structure

the result. This yields

$$Ln(y'/t) = (-1/2t_d^2)t^2 + Ln(K/t_d^2) \qquad (3.5)$$

which fits the familiar linear form $y = mx + b$.

Actual data is set up in a table form with additional calculated data points that are needed for the corresponding plot. Hypothetical data from Table IV is plotted in figure 3.3 with the best straight line fitted to the data points (determined by eye or calculation).

From this plot, Rayleigh parameters can be calculated. The slope can be used to compute development time (t_d), while the intercept (K/t_d^2), given the value of t_d just obtained, yields the value of total effort, K. Calculations for determining these values are listed with the figure.

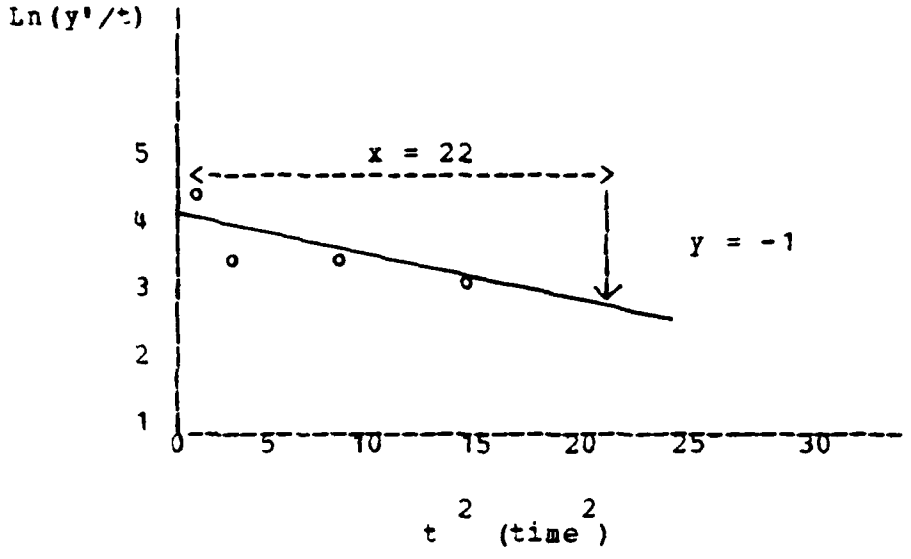
TABLE IV

Hypotheical Project Data

t	y'	y'/t	t	Ln(y'/t)
1	68	68.0	1	4.22
2	70	35.0	4	3.55
3	106	35.33	9	3.56
4	118	29.50	16	3.38

Projected management estimates can be calculated by extending the line to subsequent year points (figure 3.4). Continuing with the same example data, future man-loading predictions are made by applying the sequence of equations contained in figure 3.4.

Similiarly, resource estimation for additional outyears may be computed. As mentioned earlier, this model is an iterative procedure. Each year actual project data is added to the table. The data points are replotted and the best fit straight line is again determined. New values for the slope and intercept are found and projections are then made based on these new values.



Slope Calculations

$$\text{Slope} = \frac{y}{x} = \frac{-1}{22} = \frac{-1}{2t_d^2} = m$$

take reciprocals,

$$t_d^2 = 22$$

$$t_d^2 = 11$$

$$t_d = 3.32 \text{ years}$$

Intercept calculations

$$I = 4.0 = \text{Ln}(K/t_d^2) = b$$

$$e^{\text{Ln}(K/t_d^2)} = 4.0$$

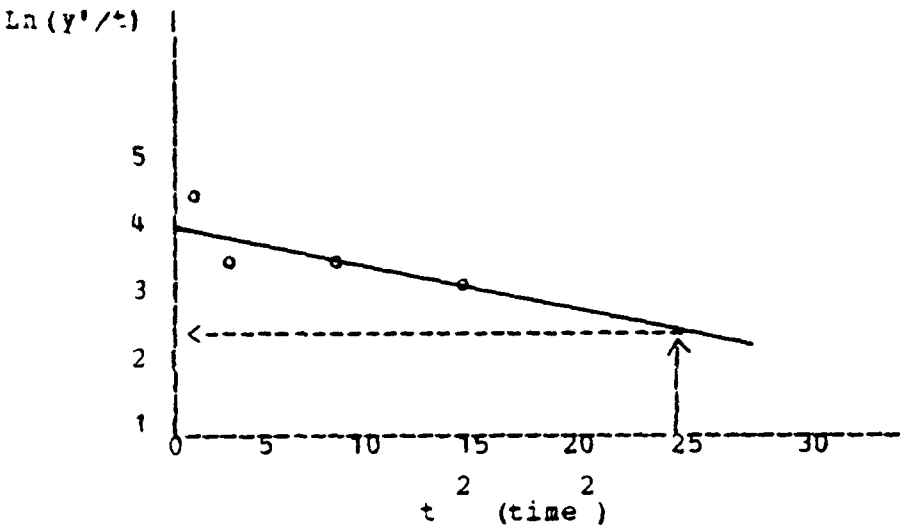
$$K/t_d^2 = 54.6$$

$$K = 54.6\ t_d^2$$

$$K = 54.6 * 11$$

$$K = 601 \text{ manyears}$$

Figure 3.3. Fitting the Best Straight Line



Projection for year 5
$t = 5$ years

$$\ln(y'/t) = 2.85$$

$$y'/5 = e^{2.85}$$

$$y' = e^{2.85}(5) = 17.29\ (5)$$

$$y'_5 = 86 \text{ people (manyear/year)}$$

Figure 3.4. Line Extension and Prediction

2. Army Macroestimating Model

a. Description

Realizing that there was a need for a simple, effective, reasonably accurate procedure for estimating and controlling resources, Army Headquarters analysts produced a comprehensive macroestimating procedure for allocating the appropriate manpower commitment to an application system at

any point in the system life cycle [106]. The procedure enables users to forecast the size of a new application software project and suggests the manloading necessary to accomplish the project workload.

Some functional estimators for the project manager include:

1) optimum man-loading over life-cycle;
2) total manpower over life-cycle;
3) cost per year;
4) risk profiles;and
5) scope of applicability.

Initial analysis of all United States Army Computer Systems Command (USACSC) systems yielded a database from which statistics have been derived that permit establishment of control limits on resource allocation at any point in the life-cycle of a system. Additionally, numerical correlation points between effort/unit time and normalized time were established for system development milestones. Using these points, the project manager can plot the project life-cycle profile of a software development effort in terms of the time that various milestones should be reached and the level of resources (manpower) that should be

applied to the system development at those points (figure 3.5).

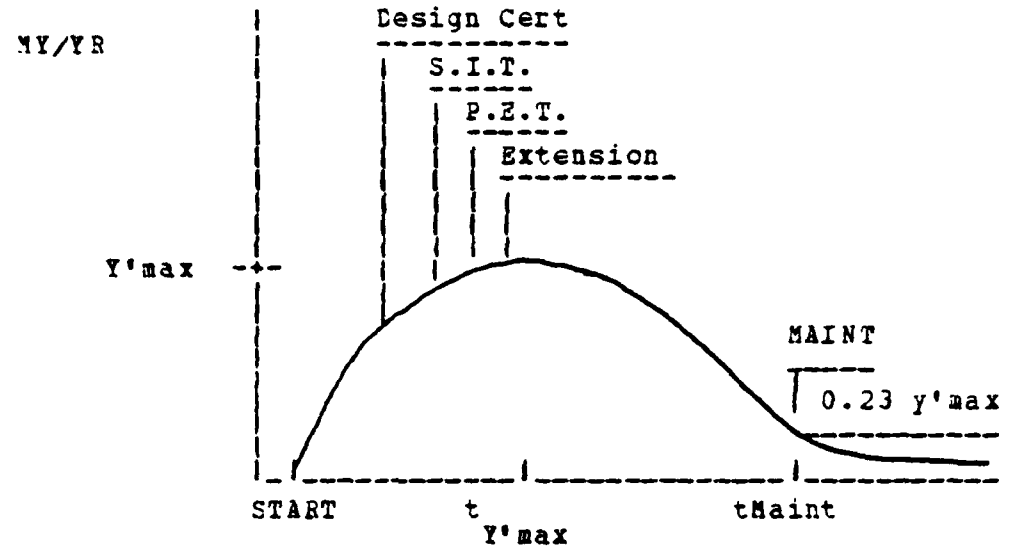
Excursions outside statistically determined control limits shown in figure 3.5 should trigger the action officer to take corrective action.

Using the mathematical relationship developed by Norden,

$$y' = 2Kate^{-at^2} \qquad (3.1)$$

step-by-step procedures were developed for estimating system variables for the following cases.

(1) Case I: System already under development (resources budgeted). Using budget data, the maximum level of manpower (y'_{max}) and the number of years to reach maximum effort ($t_{y'_{max}}$) is determined. Rather than compute the values for outyear manpower loading, Table V is used to compute the values of y' for the appropriate $t_{y'_{max}}$. By multiplying any entry opposite its time period by K, the appropriate number of manyears are obtained. The units of K and t will determine the dimensions.



DESIGN CERTIFICATION			
first	0.235		
expected		0.433	
last			0.618
SYSTEMS INTEGRATION TEST			
first	0.550		
expected		0.660	
last			0.768
PROTOTYPE EVALUATION TEST			
first	0.613		
expected		0.800	
last			0.990
EXTENSION			
first	0.613		
expected		0.930	
last			1.250
MAINTENANCE			
first	2.096		
expected		2.38	
last			2.853

NOTE 1: First and last are at five percent probability level, i.e. there is a ninety percent probability that t/tymax will lie between first and last for a particular milestone event. If not, ask questions.

NOTE 2: Tabular entries are in normalized time units.

Figure 3.5. Milestones Applied to Project Profile

TABLE V

Ordinates for Manpower Function

t	τ / y'max	1	2	3	4	5	6	7
0	a	.50	.1250	.0556	.0310	.0200	.0139	.0120
1		.60653	.22062	.10510	.06057	.03920	.02739	.02020
2		.27067	.30326	.17794	.11031	.07384	.05255	.30918
3		.03332	.24349	.20217	.14153	.10023	.07354	.05585
4		.00134	.13533	.18271	.15163	.11618	.08897	.06933
5		.00001	.05492	.13852	.14307	.12130	.09814	.07906
6			.01666	.09022	.12174	.11682	.10108	.08480
7			.00382	.05112	.09461	.10508	.09845	.08664
8			.00067	.02539	.06766	.08897	.09135	.08497
9			.00009	.01110	.04475	.07124	.08116	.08036
10			.00000	.00429	.02746	.05413	.06926	.07356
11			.00000	.00147	.01567	.03912	.05691	.06530
12				.00044	.00833	.02694	.04511	.05634
13				.00012	.00413	.01770	.03453	.04729
14				.00002	.00191	.01111	.02556	.03866
15				.00000	.00082	.00666	.08130	.03081
16				.00000	.00033	.00382	.01269	.02395
17					.00012	.00210	.00853	.01817
18					.00004	.00110	.00555	.01346
19					.00001	.00055	.00350	.00974
20					.00000	.00026	.00214	.00689

(2) Case II: New system (no resource data). Total man-years of effort and peak time for manpower loading is estimated using Bayes' theorem. [107] Based on empirical data from internal systems, a probability versus K density function was derived without regard to type of system. Further analysis determined frequency of system type and probability of occurence of each type. Using estimates based on past USACSC experiences (the average K value for all systems under development and average K for the functional type of system), initial estimates for a new new development are calculated from regression graphs. Then, by applying Bayes' theorem to average these individual estimates in the weighted probability sense yields a better estimate of K with a smaller standard deviation (i.e. better confidence in the estimate). To improve estimates and reduce uncertainty, Bayes' theorem is sucessively applied.

b. Application to Maintenance Costing

USACSC empirically determined that all of their systems reached a steady level of effort (maintenance level) on the average of 2.38 times the amount of time that was used to reach maximum effort. This relationship can be expressed as:

$$t_{maint} = 2.38t_{y'_{max}} \quad (3.6)$$

In applying this equation, a system, with maximum level of effort reached at year three, would reach a steady state at 7.14 years.

The level of effort associated with the steady state maintenance phase was empirically determined by USACSC to be twenty-three percent of y'_{max} with a ninty percent confidence interval from eight percent to thirty-eight percent of y'_{max}. At that point in the project life-cycle, when $2.38t_{y'_{max}}$ (twenty-three percent of y'_{max}) is reached, using numbers generated from the manpower equation (3.1) should be discontinued and a constant level of effort of twenty-three percent of y'_{max} should be used until the system is replaced. Figure 3.6 shows a generalized control-limit envelope of a ninety percent confidence interval for the resource level.

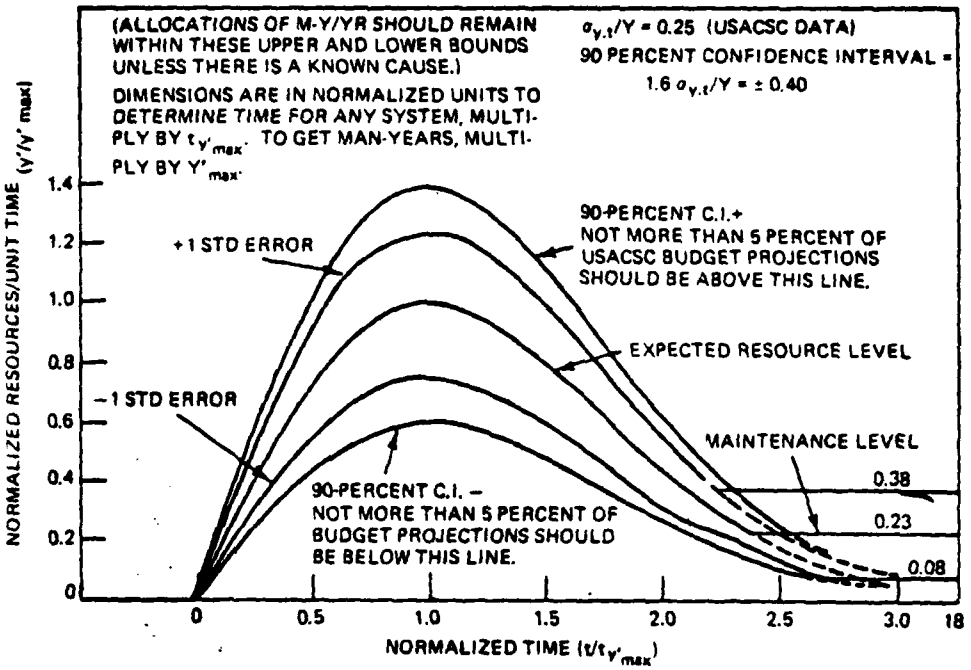


Figure 3.6. System Resource-Control Limits

B. SOFTWARE EVOLUTION MODEL

1. Lehman-Belady Model

a. Description

There have been several attempts made to assess resource allocation to achieve the repair or modification required for a single release, which is a new version of a

system. A variety of data has been collected relating to module handle rate and release interval. Based on experience in dealing with different environments, it has been suggested that development and maintenance trends exist giving rise to complexity measures. These measures, in turn, can be determined by the average number of old module handlings per new module and per fault fix, respectively.

As systems evolve over a series of releases, the ratio of changed modules to the total number of modules have been found to monotonically increase and approach unity. This ratio is an observed and directly measurable quantity which describes the system's property of resistance to change. Of importance is the indication that the number of modules involved in a system modification is likely to be proportional to the effort spent. [108, 109]

Belady and Lehman proposed a model in which activity is of three kinds: progressive (P), antigressive (A), and additional work related to system complexity (C). It was hypothesized that a balanced budget (B) implies that at any time

$$B = P + A + C. \qquad (3.7)$$

Although simple, the model captures two important aspects of evolution dynamics; the sharing of the resources between progressive and antigressive effort (where both A and C are considered antigressive) and the absorption of total budget and further growth limitation by the inevitable rise in the cost of complexity.

Increase of C activity is hypothesised to stem from neglect of A activity. Removal of resulting cumulative neglect can be accomplished only by a temporary increase in A. If the total budget, B is limited, the result is a temporary decrease in progressive activity, P. It is assumed that B, P, A, and C can be measured in cost per unit time. The cost function is expressed in the following fashion:

$$\text{Cost}(t) = \int_0^t (1 - m)\, KP(\tau)\, d\tau \qquad (3.8)$$

where

KP = inherent A activity required for each unit of P activity to prevent complexity growth;

m = management factor, the fraction of KP actually dedicated by management to A activity ($0<m<1$); and

τ = a time constant.

b. Application to Maintenance Costing

Preliminary analysis and simulation have been carried out using a non-linear differential equation model of evolution dynamics. It has been found that the model is capable of reproducing some important phenomena observed in data that can be related to observed characteristics of the system.

In figure 3.7, the simulation shows that the code production rate (the progressive element) increases to a maximum of about 225 modules per year. At the end of the first year, the complexity has increased to the point where such a production rate cannot be sustained with the budget available, since an increasing resource demand is being made by A and C activity. A balanced budget requires a reduction in P activity, which later leads to a reduction in A activity. By year six, the system has reached its limiting size with the resources available.

Although results seem promising, a great deal of work must be done before practical results in the form of an accurate predictive model can be achieved. From figure 3.7, it would seem apparent that application of control theory to modules developed earlier may result in a substantial payoff in financial terms.

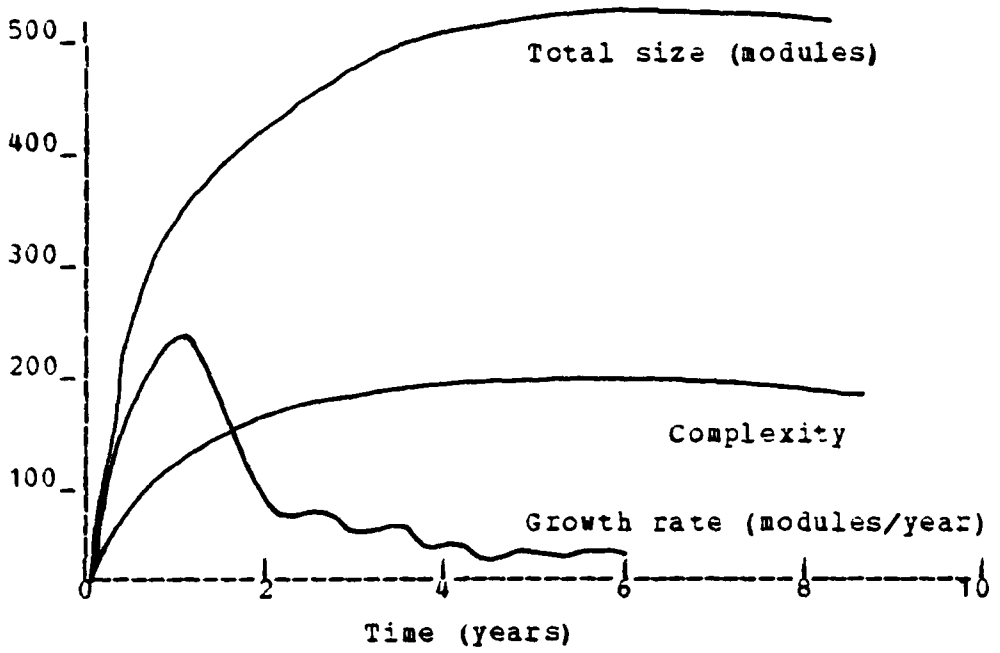


Figure 3.7. Growth Rate Simulation

2. Parr Model

a. Description

Putnam and Norden have prepared a Rayleigh curve model for the rate at which resources are consumed by software engineering projects. One of the model's main assumptions is that the initially rising work rate is due to a linear learning curve governing the "skill" available for solving problems at time t. This assumption is questionable because a linear learning curve is not theoretically supported, and the skill available on a project depends on the resources which have been applied to it [110]. Thus, this

assumption confuses intrinsic constraints on the rate at which software can be produced with management's economically governed choices on how to respond to these constraints.

Parr asserts that the rate of progress on a software project is primarily determined by dependencies among the problems which must be solved. Some problems can be solved in parallel whereas others can only be handled sequentially. Let W(t) be the number of problems which have been solved at time t and V(t) be the number of visible unsolved problems at time t which can be solved (i.e., all earlier required problems solved). When a problem is solved, W(t) increases by 1; V(t), however, may increase or decrease depending on whether or not the problem solved makes new problems invisible/solvable. It is reasonable to assume that problems solved early in the project will lead to more unsolved problems, and that those solved later will have a higher probability of not making new unsolved problems visible. A crude approximation to this is to assume that the probability of a solved problem not generating more unsolved problems is linearly proportional to the number of problems solved. [111]

How does the above relate to the rate at which development programs can be made? Clearly, management can reduce the rate of progress by supplying inadequate resources. There is also an upper bound to the amount of effort which can be usefully applied. Rapid progress using large amounts of input resources is possible only when there is scope for solving problems in parallel. In practice, a different programmer (or possibly a team of programmers) can be assigned to each separate visible unsolved problem. This suggests that the rate at which useful work can be applied is proportional to V(t), and that with this "optional" input effort, steps in the development will be achieved at a rate proportional to V(T).

Whereas the Rayleigh model proposes that the rate of progress will be proportional to the skill level and number of problems remaining, the above has argued that it is proportional to the visible unsolved problem set. A mathematical expression yields:

$$V(t) = (1/4)\ \mathrm{sech}^2\ ((t + c_3)/2) \qquad (3.9)$$

a hyperbolic function symetric in t with an integration constant c_3; while the Rayleigh function is:

$$V(t) = y' = te^{-t^2/2} \qquad (3.10)$$

The $sech^2$ model closely resembles the Rayleigh model in the latter half of the curve, but the front tail is positive rather than zero like the Rayleigh (figure 3.8). Thus, in the $sech^2$ model, projects do not have well-defined starting points. This accounts for work done prior to the official project starting date.

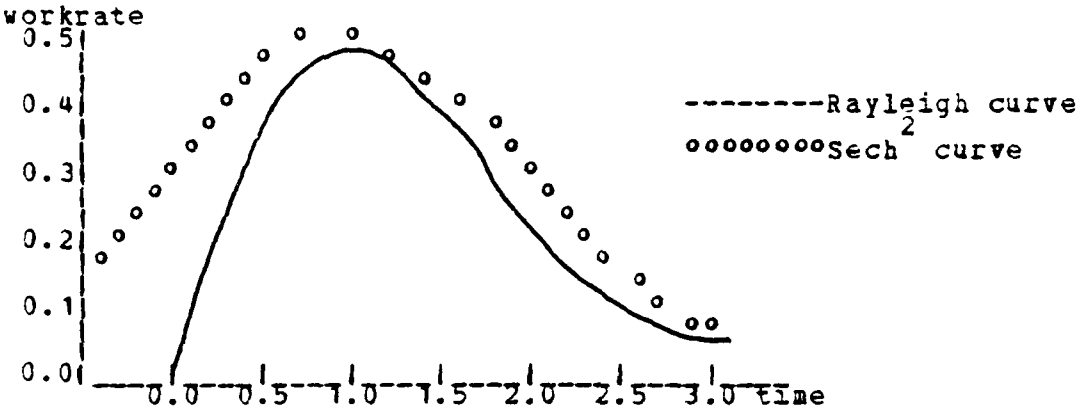


Figure 3.8. Sech Curve

One of the principles of software programming is that decisions initially made should be high-level structured ones which identify components for subsequent refinement. Increasing the complexity of the initial decisions in this manner is equivalent to varying the distribution of the probability of a solved problem generating unsolved ones. By modifying the assumption that this probability is linear, a modified workrate function can be derived:

$$V(t) = \frac{A \exp(-2\alpha t)}{(1 + A \exp(-2\alpha t))^{3/2}} \qquad (3.11)$$

Thus, it may be seen that whereas the Rayleigh model of software development proposes that the rate of progress will be proportional to the skill level and number of problems remaining, this section has argued that it is proportional to the size of the number of the visible unsolved node set.

Results obtained from the proposed model are similiar to the Rayleigh model, except that account is taken of work contributing to a project which precedes its official starting date. The proposed model has been shown to be sufficiently determined for it to be possible to account for the effect of different programming methodologies on the natural work associated with the project.

b. Application to Maintenance Costing

Parr suggests that exhaustion of the problem space is the main cause for decrease in maintenance effort at the end of the project profile curve. In addition, the structure of the software product achieved during the development could affect the project work profile. While applications to maintenance costing have been addressed in

concept only, implications are that integration techniques for determining the area under the curve at a specific time pe- riod will produce results similar to those obtained by using Putnam's model.

C. CHAPTER SUMMARY

With the intent to gain management control of predicting maintenance costs, various software cost estimating methods and philosophies derived from observing trends and patterns in the development cycle are being extended to encompass total system costs. Supportive evidence for the accuracy of the models discussed herein is contained, for the most part, in software life-cycle simulations. It is anticipated, however, that the acute interest and increased awareness shown in the resource investments attributed to software maintenance will be viewed more critically. Although lacking in substantial proof for predictive validity, these models serve as stepping stones in producing a composite model for tracking maintenance costs.

IV. MANAGING SOFTWARE MAINTENANCE COSTS

Previous chapters have addressed topics that are being critically examined for their impact on the software maintenance phase. They also discussed the application of current development software cost estimation techniques for obtaining maintenance costs. The focus of this chapter will be the presentation of a method for arriving at a well-structured view of the management of the maintenance phase of the software life-cycle. While a mathmatical model which accurately explains the phenomena of the maintenance phase still remains elusive, a planning and control model has been developed to aid project managers. The structure of the model embodies two distinct concepts:

1) a planning concept - development of the management strategy to cement the perceptions of the maintenance issue;

2) a control concept - procedural analysis for estimating the maintenance manloading requirements.

Subsequent sections will address application of each aspect of the model in depth.

A. PLANNING CONCEPT

1. Project Management

Primary responsibility for development of a management strategy belongs to the project manager designated to manage the system plan. As project manager, one must initially determine and define the maintenance requirement of the mission profile for the system that is to be designed (i.e. built-in maintainability).

Factors which must be considered early in the formulation of a maintenance plan include:

1) Probability of change in requirements. While it may be impossible to define adequately the complete requirements for a large program, viewing the type of system application (business, scientific, command and control) and utilization rate will serve as indicators for the amount of flexibility to be considered in the system design.

2) Software performance requirements. Again, application type is the dictating force for analyzing this factor.

3) Hardware life-cycle. In planning for software maintenance, the interaction of the hardware and software life-cycle must be taken into account.

2. Objectives of the Maintenance Concept

Derivation of maintainability requirements from the description of the operational requirements provides the support planning criteria on which to base the maintenance concepts appropriate to the maintainability requirement. The maintenance concept, which basically defines criteria

governing the scope and methods applicable at each echelon of maintenance, attempts to satisfy the quantitative maintainability requirement derived for the system and the planned support environment within which the system will operate. Early development of the appropriate maintenance concept will provide a definitive and uniform basis for accomplishing the system design and support planning tasks.

3. Establishing the Maintenance Policies.

System effectiveness is jointly dependent on several parameters, of which performance characteristics, system reliability, and operational availability appear to be the most critical. In effect, these parameters set baseline requirements or constraints which may have impact upon the design process depending upon the maintenance policy that has been established. While a boundless number of policy variations may exist, the following four categories identify the range of policy choices. The basic distinction among these four categories lies in the amount of resources invested over time and the cumulative benefit received over time.

a. Category I - No Management Control

A steady maintenance effort is applied with no attempt for configuration control which is ensuring that a master copy of all operational software is maintained. Complexity of the program will eventually reach the point where locating errors and/or making changes becomes exceptionally difficult. Gradually, the program becomes less useful until it must be discarded and a new program developed. This policy may prove to be cost effective for situations where it is known that the nature of the application will limit the useful life of the program.

b. Category II - Permanent Support Level with Periodic Redevelopment

As in Category I, a steady level of maintenance support is provided by a permanent workforce. Redevelopment or a new release can be planned for at regular intervals or in response to a specific quantity of change requests.

c. Category III - Error Repair with Major Changes

Manpower support is set at the level needed to correct program bugs. External programming support would be required for making major changes.

d. Category IV - Error Repair Only with Periodic Redesign

As in Category III, manpower is set at the level needed to correct an unacceptable design error or program bug. Change requests are used in establishing specifications for subsequent design of a new program.

4. Management Structure

Since the level of repair policy must be compatable with the maintainability requirement, the maintenance concept must be defined for each management level of maintenance established. Beginning with the lowest level of users, maintenance concepts are implemented with subsequent policies for higher management levels developed to support the user level concept. To illustrate, maintenance may be divided into three echelons as discussed below and shown in figure 4.1.

1) User level. Maintenance may be restricted to failure reports and system restarts.

2) Organizational level. Technicians perform corrective maintenance. Tasks performed would include location of fault, module repair, and testing.

3) Contractor level. Maintenance performed at this level may be used to supplement (augmented support) or to replace (sustained support) the organizational level support.

Activity	User Level	Organization Level	Contract Level
Where performed	Remote or local site	Facility having project cognizance	Agency and/or contract facilities as required
Who performs	Maintenance personnel	Maintenance division or support team	Contract personnel
Maintenance action	Restore system to operational status	Locate module errors; repair and return to user	Locate module errors; repair and return to user
Maintenance tasks	Inspection and restarts minor repairs and adjustments submit change requests	Module repair; major coding modifications, testing	Complex repairs, modifications; major coding; rebuilds

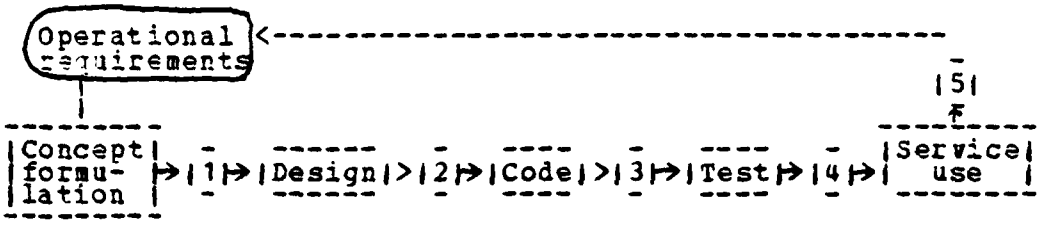
Figure 4.1. Maintenance Levels

5. System Life-cycle Objectives

Utimately, the maintenance objective is to achieve the required level of maintainability in delivered systems with an optimum balance between resource support requirements and potential life-cycle costs. In order to meet this objective, it is necessary to begin the system life-cycle with the appropriate conceptual approach. As the software product passes through several distinct phases in its evolution, maintenance prospects can be enhanced if adequate attention is taken in each phase.

Figure 4.2 depicts the life-cycle as a simple phase-to-phase flow diagram, joined by critical transition points where it can be ascertained that the required maintainability objective has been achieved before transition to the next phase. These transition points are denoted in the figure as major achievement milestones. Each phase comprises several areas of management endeavor in which the consideration of the system maintainability is essential to the attainment of milestone objectives. The software product is re-examined at each milestone to determine the future course based on progress up to that point. As each milestone objective is met, maintainability becomes progressively more tangible as a built-in feature of design. Maintainability milestone requirements are summarized in the figure.

Milestone criteria can best be satisfied by systematic application of approved procedures in the performance of evaluation, management, and control tasks which are geared directly to specfic objectives of individual milestones in the life-cycle. A basic approach to maintainability achievement as an evolutionary phase-to-phase 'growth' process is shown in figure 4.3.



Concept Formulation Phase Milestone Criteria. Maintainability requirements derived; maintenance concept established; maintainability documented in system specifications; maintainability milestones and task requirements documented.

(1) Proceed to design phase.

Design Phase Milestone Criteria. Maintainability design approach and maintenance concept optimized by tradeoff and conformance to specified requirements and economic considerations; maintainability requirements and milestone criteria updated.

(2) Proceed to code phase.

Code Phase Milestone Criteria. Conformance to specified maintainability requirements and maintenance concepts verified by evaluation; maintenance control procedures defined in support documentation.

(3) Proceed to test phase.

Test Phase Milestone Criteria. Maintainability degradation factors verified by test and evalution; maintenance concepts, repair policies, and maintenance procedures are verfied.

(4) Software product is approved for delivery.

Service Use Phase Milestone Criteria. Maintainability characteristics, maintenance prodecures, and support costs determined by periodic assessment of management data; problem areas identified for correction.

(5) Initiate change request, product enhancement or new development; repeat life-cycle.

Figure 4.2. Maintenance Milestones in the System Life-cycle

B. CONTROL CONCEPT

1. Objective of Maintenance Control

The objective of this thesis was to develop a methodology for arriving at a good prediction of pure

TASK AREA	LIFE-CYCLE PHASE					TASK REQUIREMENTS
	CF	D	C	T	S	
Determine Maintainability Requirements	*	*				° Establish M policies ° Derive M requirements ° Optimize M to reliability, availibility, and supportability ° Evolve conceptual design criteria
Specify maintainability requirements and milestone criteria	*	*	*	*		° Define M requirements ° Define M milestone criteria and task requirements in program documentation ° Specifically outline foregoing requirements in contractual documents
Achieve specified maintainability in design		*	*	*		° Identify and define M problems and critical areas ° Integrate M enhancement into design ° Verify design conformance to specific requirements ° Review impact of proposed changes on M design characteristics
Show adequacy of maintainability in development			*	*		° Prepare detailed plans for maintenance test ° Demonstrate adequacy of maintenance manuals
Achieve optimum maintenance support	*	*	*	*	*	° Develop maintainence plan, repair policies, and procedures ° Develop maintenance training program ° Prepare contractor support plan
Evaluate maintainability at service level					*	° Verify conformance to M requirements under service conditions ° Verify adequacy of maintenance support manuals ° Evaluate skill requirements and adequacy of training program

CF = Concept Formulation D = Design
C = Code T = Test
S = Service Use

Figure 4.3. Maintenance Tasks in the System Life-cycle

maintenance costs. Determining the requirements for pure maintenance is considered valuable in that

1) estimates can be calculated of the manloading necessary to form a maintenance support team which is composed of either in-house or contract augmentation;

2) projections can be made for outyear maintenance support and availability of manpower resources for developmental work.

With future research, the application of this model may be extended to any software project; however, access and availability of data precluded analysis of small and medium sized projects. Only data from major projects was analyzed for developing a computational algorithm.

In executing the computational algorithm, both macro (system) and micro (functional area component) techniques are used concurrently to increase the validity of the estimates. An implicit assumption worth noting is that each method should provide reasonably close estimates for the same project. The macro technique, of course, is based on total system characteristics and will provide the gross manning requirements directly. Alternately, from the micro technique, summation of the decomposed functional areas will yield the gross manning for the total system.

2. Model Deriviation

The data under study was taken from a large-scale project reported by USACSC [112] and unpublished data from the IBM Space Shuttle Program [113].

a. Macro Technique

Using the Rayleigh curve parameters derived by Norden and Putnam [114, 115], a method was constructed for obtaining total system maintenance requirements applicable to the established management strategy. In his early work, Norden made note of the fact that the Rayleigh curve of a project profile has a point of inflection at which the decrease in utilized manpower slows down in the descending portion of the curve,

$$t_{ip} = \left(\frac{3}{2a}\right)^{1/2} \tag{4.1}$$

where

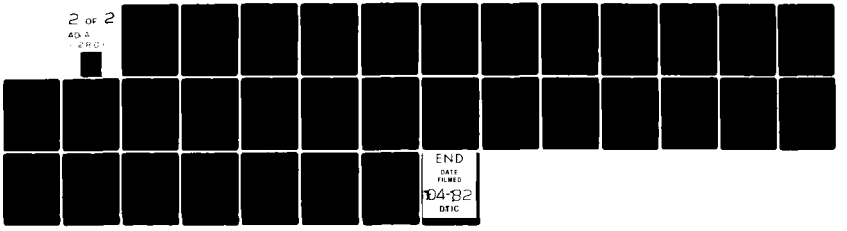
t_{ip} = the inflection point of the project curve

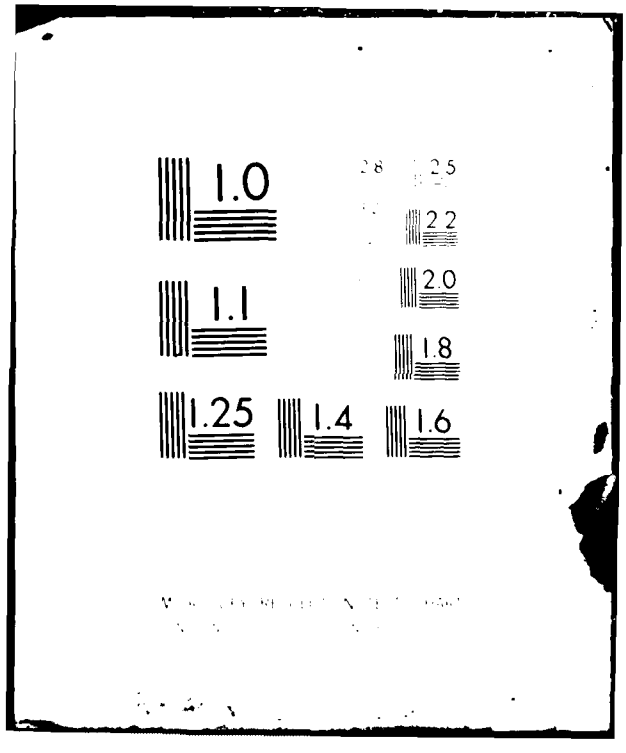
a = the shape parameter or spread of the curve.

The point of inflection may have more significance than originally recognized. If it can be shown that the level of effort for the maintenance phase reaches a maximum at this point, the manloading estimate calculated from

AD-A112 801 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 14/1
DYNAMIC PLANNING AND CONTROL OF SOFTWARE MAINTENANCE: A FISCAL --ETC(U)
DEC 81 J F GREEN, B F SELBY

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
2.2
2.0
1.8

this point can be used as the upper bound for maintenance support. In essence, the current model suggests a new mechanism for determining the level of maintenance support required. Gained from the model is the benefit of relating the work profile more directly to the intrinsic structure of the project profile.

To simplify calculations, the project profile is normalized with respect to t_d and y'_{max} as shown in figure 4.4. Total life-cycle (K), has a normalized value of 1. Based on this assumption and using Rayleigh curve relationships, it can be shown empirically that the peak of maintenance effort occurs at the inflection point.

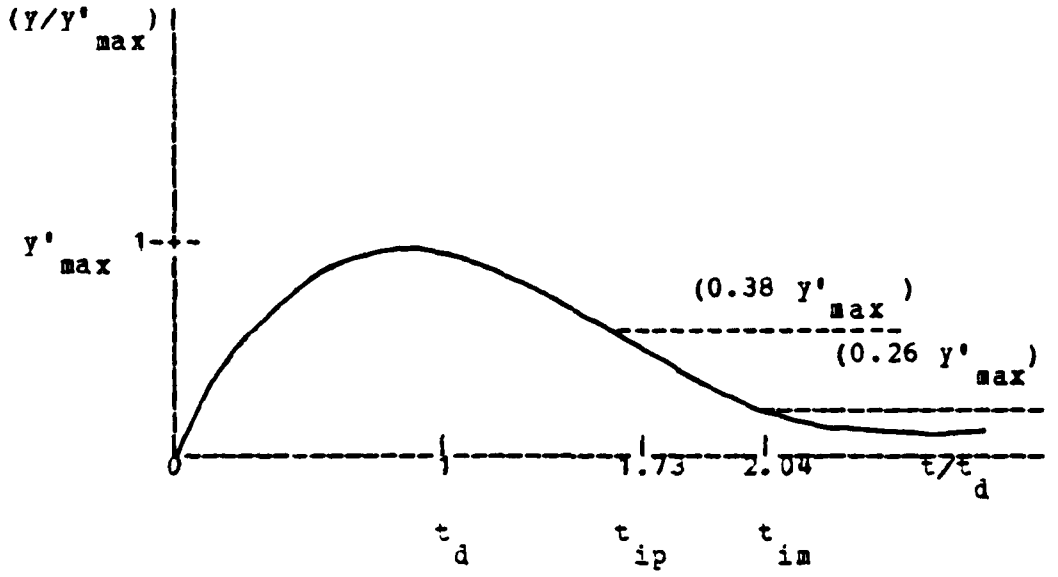


Figure 4.4. Normalized Rayleigh Curve

From the normalized curve values, the shape parameter is found with the following relationship:

$$a = \frac{1}{2t_d^2} = 0.5 \qquad (4.2)$$

Substituting this value of a in the following equation, the inflection point (t_{ip}) of the project profile is obtained.

$$t_{ip} = \left(\frac{3}{2a}\right)^{1/2} = 1.73 \text{ years.} \qquad (4.3)$$

Manloading requirements at time t_{ip} ($y'_{1.73}$) can be shown mathematically to be equal to the maximum manload (y'_{t_m}) which occurs at the peak (t_m) of the maintenance phase profile. Stated in the format of a mathmatical equation, this equality has the form

(project inflection point manning) = (maximum maintenance phase manning)

or $$y'_{t_{ip}} = y'_{t_m} \qquad (4.4)$$

Substituting normalized values into the Rayleigh (manpower) equation

$$y'_{(1.73)} = 2(1)(.05)(1.73)e^{(-.05)(1.73)^2} \qquad (4.5)$$

$$y'_{(1.73)} = 0.38 \text{ manyears.} \qquad (4.6)$$

In order to calculate maximum manning for the maintenance phase, parameters for the maintenance curve must be defined. Actual time elapsed between the beginning of the maintenance phase (t_o) and the maximum level (t_m) is computed using empirical data recorded by USACSC. Results from the USACSC research indicate that the maintenance phase, which accounts for twenty percent of the total life-cycle manpower (K), begins at approximately 1.3 years normalized time. With this estimation, actual time elapsed (t_e) can be found by

$$t_e = t_m - t_o = 0.43 \text{ years.} \tag{4.7}$$

The spread of the maintenance curve (a_m) is determined by subtituting the elapsed time value into the already familiar equation

$$a_m = \frac{1}{2(t_e)^2} = 2.71 \tag{4.8}$$

The value obtained for the shape parameter suggests a curve having a wide spread, an expected characteristic of the maintenance phase profile curve. Computation of the max manloading for the maintenance phase from the basic manpower equation gives

$$y'_{(t_m)} = 2(.2K)(2.71)(.43)e^{(-2.71)(.43)^2} \quad (4.9)$$

$$y'_{(t_m)} = 0.38 = y'_{(t_{ip})} \quad (4.10)$$

With y'_{t_m} defined to be the upper boundary for the maintenance effort, another boundary can be identified as the lower limit for maintenance effort. By determining the value of the inflection point of the maintenance curve (t_{im}), a minimum support level can be found from

$$t_{im} = \left(\frac{3}{2a_m}\right)^{1/2} = .74 \text{ years.} \quad (4.11)$$

Converting this time to normalized time (t_n)

$$t_n = t_o + t_{im} \quad (4.12)$$

$$t_n = 1.3 + .74 = 2.04 \text{ years} \quad (4.13)$$

and substituting this value in the manpower equation yields

$$y'_{(2.04)} = .26 \text{ manyears.} \quad (4.14)$$

The manpower loading calculated for the inflection point of the maintenance phase (t_{im}) closely approximates the value identified by USACSC as the steady state level of effort. Establishing maintenance at the minimum level can be interpreted as a Category IV policy.

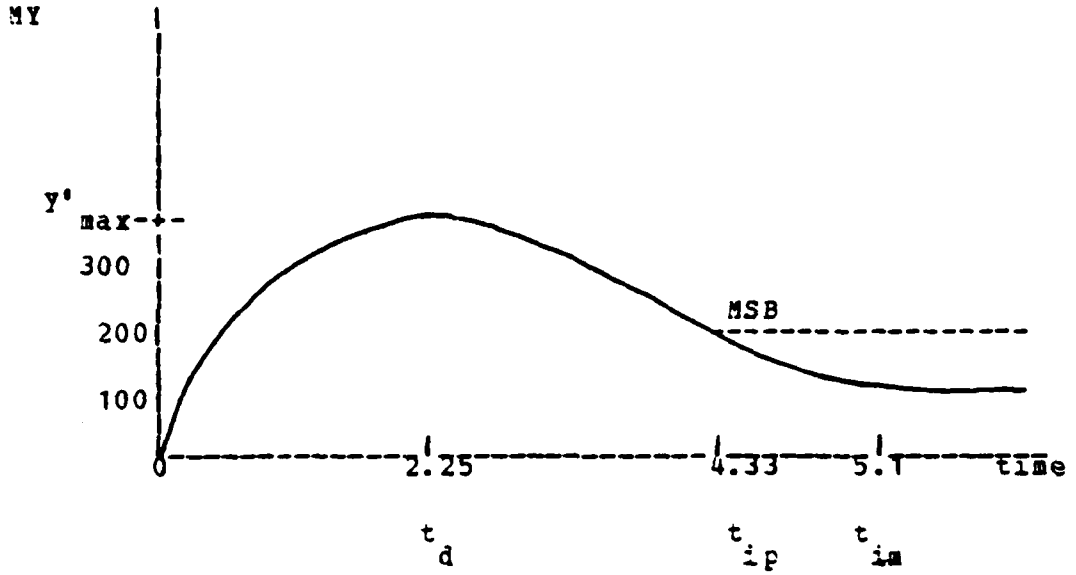
b. Micro Technique

Decomposition, more commonly referred to as the work breakdown structure (WBS) method, has been a predominate methodology for estimating manning resources. A system is considered to contain subsystems which are further divided into smaller hierarchial structures until the smallest programming element is reached. Once the functional areas are defined, characteristics (complexity, productivity, error rate, etc.) of each must be reviewed to determine the level of effort needed for maintenance. Appendix A contains an example of a micro-estimating methodology along with the sample data used.

3. Sample Application

a. Sample Data

Data used for this sample application of the control concept was provided by IBM Federal Systems Space Shuttle Program [116]. The raw manning data is provided in Appendix A. The remainder of this section is a step-by-step example of the computional algorithm which implements the control concept of the proposed model.



Project Curve Parameters	Maintenance Support Level
$t_d = 2.5$	(1) Macro technique
$K = 1343$	$t_{ip} = 4.33$
$a = .08$	$y'_{(4.33)} = 207$ manyears
$y'_{max} = 325$	$t_{im} = 5.1$ years
NOTE 1: MSB = maintenance support boundary	$y'_{(5.1)} = 137$ manyears
Boundary level established from analysis of macrc and micro estimations.	(2) Micro technique
	y_c = component manning
	$\sum y_c = 195$ manyears
	(refer to Appendix A)

Figure 4.5. Plotted Sample Data

b. Computaticnal Algorithm

Step 1. Fit the actual budget data to a Rayleigh curve. Figure 4.5 shows plotted data for the Space Shuttle program.

Step 2. Determine maintenance support boundary lines by calculating the inflection points of both the project profile and maintenance phase curve.

Step 3. Determine support level requirements using micro-estimating techniques.

Step 4. Compare values obtained from macro and micro methods. Analyze the differences from an economic standpoint based cn management policy.

Step 5. Predict outyear budget requirements for maintenance/new develcpment contingent on management policy.

c. Management Applications

Althcugh the results shown here relate to only one set of data, they are encouraging in the support they give the model. The model presented in this thesis could provide a direct means to evaluate the impact of current and future management practices on the life-cycle cost of the software system. The idea of the development of a maintenance strategy coupled with the use of the computational

algorithm provides the project manager with some powerful management tools. While additional research is warranted, it is purported that application of the model will prove enlighting in the following respects.

(1) Determining Maintenance Support Level. Preliminary estimates obtained from inflection point predictors may be used as a starting point for planning workforce requirements to be drawn from internal assets. Likewise, if external or contracted support must be procured, evaluations of submitted bid proposals will be necessary. Although yet unproven statistically for accuracy, the inflection point predictors appear to define maximum (t_{ip}) and minimum (t_{im}) boundaries for maintenance levels.

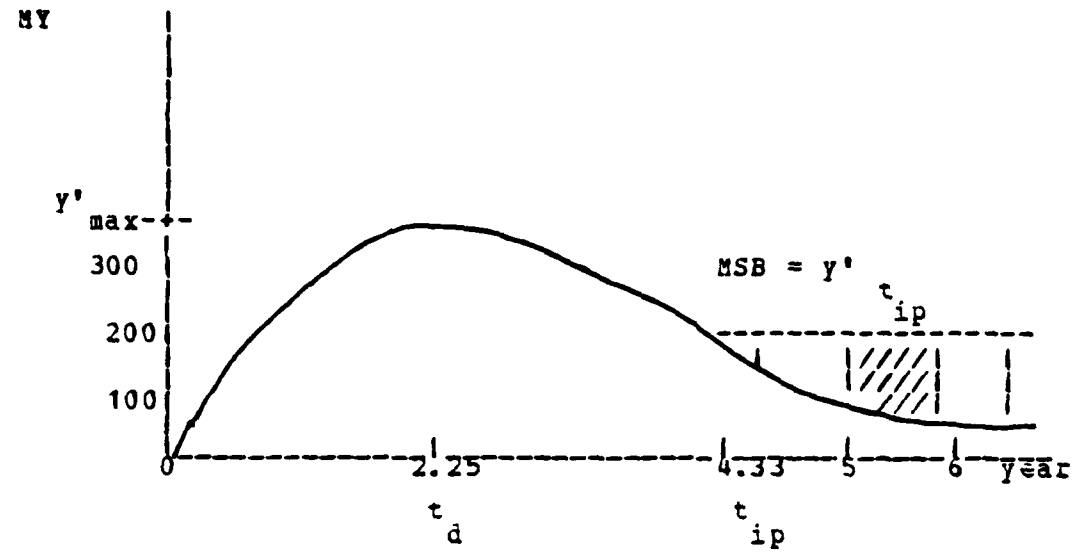
In accordance with the type of maintenance strategy chosen, a maintenance level boundary can be selected. For example, if a Category IV policy is selected, manpower needs would approach the minimum boundary. On the other hand, a Category I policy would require resources approximated by the maximum level. With these boundaries to serve as guidelines, contract proposals can be viewed more critically.

(2) Forecasting Resource Distribution. Whether an internal or external workforce is used, planning and budgeting estimates of manloading are usually projected for discrete timeframes. During the maintenance phase of a viable project, the workforce in terms of total number remains stable; however, the work distribution or functional roles of personnel may change (i.e. programmers may shift from maintenance work to development work). Within governmental agencies, this stability may be attributed to fixed contract levels or established manning levels, neither of which can be easily changed. Therefore, the management problem becomes one consisting not only of how many personnel are needed, but also how can assets best be utilized.

In light of the fact that the users have changing requirements, the issue of workforce allocations for new research and development must be considered. Based on the Rayleigh curve characteristics for a specific project and using a fixed support level environment, approximate values for workload distribution can be calculated.

By method of integration, the proportion of the total support level force that will be dedicated to pure maintenance and/or new development in future timeframes can

be calcalated. Figure 4.6 illustrates this point using the sample data.



$$\int_{t_1}^{t_2} MSB - 2Kate^{-at^2}\,dt$$

$$= 207t\,\Big|_5^6 - 1343e^{-at^2}\,\Big|_5^6$$

= 101 MY (resources available for new development)

Note 1:
MSB = maintenance support boundary. In this example, the boundary is established at the project profile inflection point. Alternately, the boundary would be established to indicate the manning level of the maintenance workforce.

Figure 4.6. Forecasting Future Requirements

Maintenance information gained with this oversight method is twofold. The separation of development work (enhancements, additions, new design) from maintenance

work (debugging, design error correction) is accomplished, thereby allowing for better interpretation of the project investment. The comparison of the relative proportion of maintenance manning versus development manning for reviewing project viability can also be made. This concept will be discussed more fully in the next section.

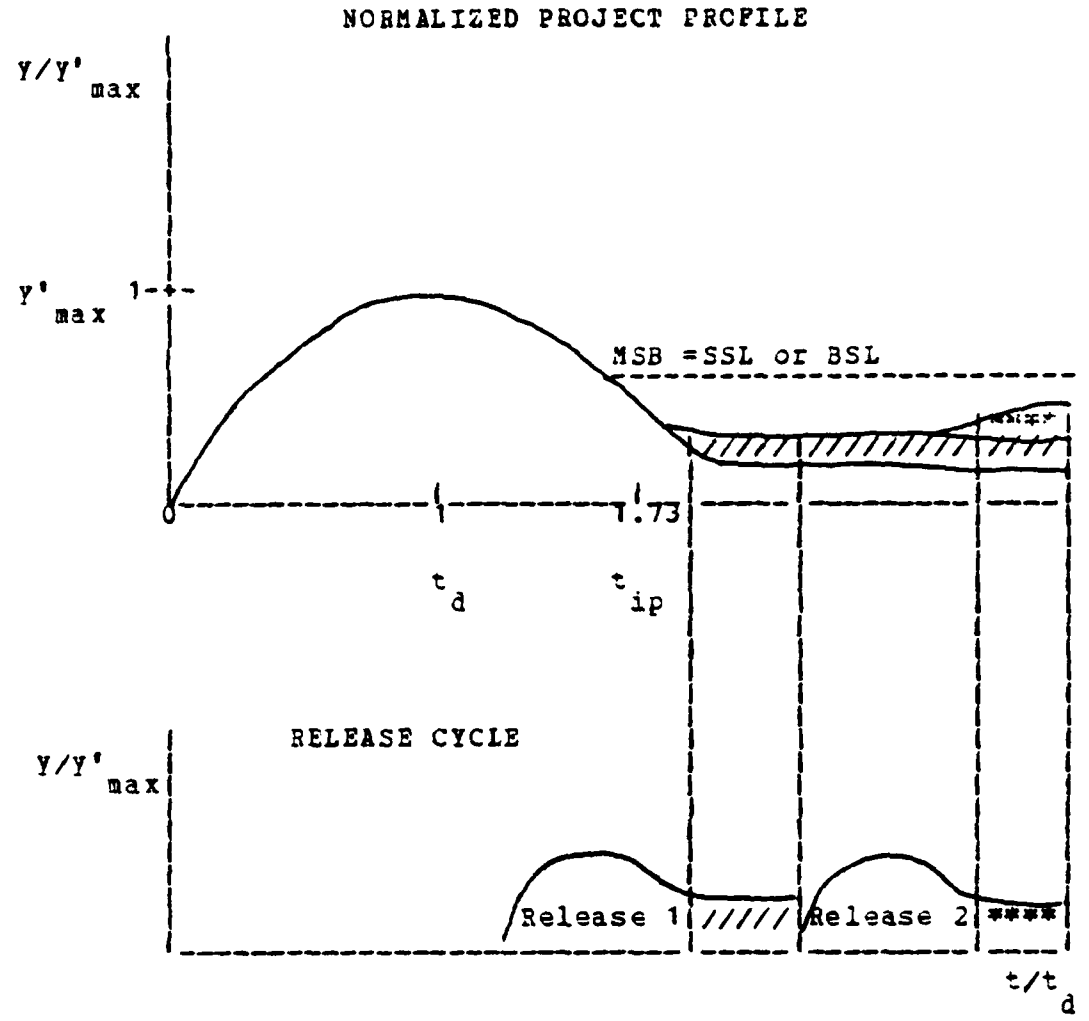
(3) Monitoring Configuration Control. A paucity of available data prevents the comparison of actual and predicted manpower that is required during the maintenance phase. The assumption that the maintenance tail is flat or reaches a steady state seemingly arose from this lack of information. It is the authors' contention that new releases of a software product may, in fact, cause increases in the maintenance tail over time.

Lehman and Belady's [117, 118] research, discussed in chapter 3, gives strong indications that subsequent releases for a software product increase complexity and the amount of antigressive (maintenance) work that is required for the total system. Two inherent characteristics of the software product directly affected by a new release are the system configuration and the size of the system problem space. From a project profile view, the time period

when these new releases occur is during the maintenance phase. With the assertion that the work allocated to the completion of a new release must be considered as a phase within the project life-cycle, the increase in maintenance costs can be explained.

As the diagram in figure 4.7 indicates, the changes induced by the release phase will cause the level of maintenance to increase. Unless carefully monitored, each new release may cause an increase in the maintenance requirements until the original maximum maintenance support level is reached or exceeded. When this occurs, management is forced to make a cost-benefit assessment of the software system.

Using the concepts introduced earlier (inflection point predictors and resource distribution forecasting), maintenance saturation of the software system can be detected. The support line obtained from the inflection point predicter (t_{ip}) serves as a guideline for total system saturation. Management policy sets forth limits for corresponding maintenance and/or development expenditures which establishs a budget saturation level. These saturation levels may be equal or different. In an attempt to preclude



Note 1:

MSB = maintenance support boundary

SSL = system saturation level, equivalent to curve inflection point where any maintenance above this level would be considered antigressive

BSL = budget saturation level, established by management where possible values may be:
BSL = SSL
BSL < SSL
BSL > SSL

Figure 4.7. New Release Effect on Maintenance Level

excessive maintenance costs, the saturation level viewed as dominant is used to trigger management's attention toward a system rebuild. For the subsequent rebuilt system, a new Rayleigh curve is plotted and a new cycle of planning and control begins.

C. CHAPTER SUMMARY

Presented in this chapter is a bilevel model for managing software maintenance costs. The model, composed of both a planning concept and a control concept, suggests that the creation of a management strategy will have far-reaching effects in the system total life-cycle costs. Used concurrently, the two model concepts allow for smoother translation of maintenance objectives between the strategic planning level and the operational control level.

V. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

Presented in this chapter is a summary of the thesis, general conclusions, and recommendations for further study.

A. SUMMARY

Various methodologies and system factors relating to software cost accounting have been reviewed in an attempt to develop a cost model for the prediction of pure maintenance costs. The distinction between development costs and maintenance costs is considered necessary in order to present a realistic picture of the annual expenditures within a given budget constraint. Without a refined separation of these two cost entities, budget control is a more difficult task.

Beginning with a broad background of what maintenance is and is not, Chapter One uncovers the paradox that exists in obtaining a consensus for a common working definition of maintenance. Different schools of thought within the military and civilian research fields have produced inconsisent results when citing the proportion of the the life-cycle costs attributable to total system costs. This inconsistency may be due, in part, to the range of cost types (new

development, pure maintenance, other administrative support activities, etc.) that exist during the maintenance phase. While some researchers may view each cost type individually, others consider the maintenance costs to be an aggregate of all expenditures during the maintenance phase.

In Chapter Two, an overview of the extrinsic and intrinsic charactistics of a software system which create the maintenance setting is provided. It is apparent from the detailed discussion of the more salient concepts that the maintenance issue is not only complicated, but also still somewhat elusive. While these concepts have been useful in explaining system characteristics and predicting future behavior, they fail to produce a means for direct translation to a monetary value.

Although no cost estimation technique adaptable for management use has been developed solely for predicting maintenance costs, application of software cost estimating schemes originally intended to evaluate the development phase have been extended to include the maintenance phase. Chapter Three is devoted to a review of various models that have been suggested as appropriate for addressing the maintenance cost uncertainties. The models two avenues for approaching the issue:

1) a total system concept using the Norden-Rayleigh curve, and

2) a dynamic system philosophy using software evolution analysis.

Current unavailability of a basic method for adequately determining miantenance expenditures and the increasing concern of DOD for the exorbitant funding required to sustain software system operations inspired the authors to develop a flexible management model. Chapter Four elucidates a planning and control model which can provide project managers with additional information to assist in budget planning and decision making. This model proffers four maintenance strategies which may be used in conjuction with calculated maximum and minimum maintenance level support boundaries specific to the project profile.

B. CONCLUSION

While an abundance of research in scftware economics and software engineering exists, very little has been done that relates to the maintenance phase of the software life-cycle. As a result, there is an obvious lack of raw data available to analyze the proposed model for validity and sensitivity.

With additional research, it is believed that the model presented in this thesis will provide a direct means to

evaluate the impact of current and future management practices on the life-cycle costs of software systems. The combined use of the simple macroestimating and microestimating techniques allows the manager to look at the maintenance problem from different perspectives while increasing the confidence in the projected maintenance costs. Additionally, the computation of minimum and maximum levels of effort for a specific project leads to further diminution of the problem when management has established a particular maintenance strategy.

C. RECOMMENDATIONS

It is recommended that additional work within DOD be undertaken to further the research objectives of software maintenance costs and that this work include the following actions:

1) adoption of a standard definition that will distinguish between maintenance costs and costs incurred in the maintenance life-cycle phase;

2) institution of longitudinal research by software support facilities to collect maintenance data to be used in the development of management tools with improved capability;

3) investigation of the usage of additional prediction tools to obtain a more complete view of the domain of software behavior during the maintenance phase; and

4) analysis of empirical data to prove or disprove the following statement: The more over-budget and behind schedule that a project is delivered, the higher should be the prediction of errors detected in the maintenance phase.

APPENDIX A

Contained in the following text is a partial summary of a microestimating technique (Matrix Estimation Process) obtained from IBM Federal Systems Division, Houston, Texas.

MATRIX METHOD

Definition: The Matrix Method is a systematic procedure which can be used to delineate elements of a software project and map them against associated cost elements to arrive at a project estimate.

Things that can be accomplished:

- Lay out project elements
- Stepwise refine the elements
- Estimate the elements
- Subtotal the estimates by grouping the elements
- Total up the group estimates
- Refine total estimate

Use of the Matrix Method

1. Determine functional elements of project.
2. Quantify Maintenance needs based on :

   Level = Function Size / ((Productivity) (Complexity) (Factor) .

3. Consider critical skills, operations support, and management and support.
4. Summarize for project.
5. Plot with Rayleigh curve.

Matrix Estimation Process

1. Lay out a table of functional areas of code, requirements areas, test areas, or functions to be performed. Using the following formulas, calculate the level required to maintain each functional area or function:

   Applications Level = (FW Size)/((154) (2) (12) (2))
   FCOS Level = (FW Size)/((100) (2) (12) (2))
   SDL Level = (K Line Size)/((15) (2))
   Computer Resource = (FEID Hrs Wk)/ (28)

   GN&C Verfi. Level = (Number Test Cases)/((30) (2))
   SSW Verif. Level = (Number Test Cases)/(5) (2)
   SM/PL Verif. Level = (Number Test Cases)/((20) (2)
   Perf. Verif. Level = (Number Test Cases)/((5) (2)

   All Others:
   Level = (Development or Support Level)/(2)
   T&O Level = Requested support level per site

2. Look at critical skills to see if each functional area is adequately covered.

3. Estimate error rate and rate of change to see if level should be altered.

MATRIX ESTIMATE SUMMARY*

Area	Size	Maint. Level	CPN & Support	M&S	Total
AASD	272918 FW	42.0	12.0	10.0	64.0
CON/QA	-----	5.0	---	1.0	6.0
SEC. SUPP	-----	11.0	---	---	11.0
ASVO	1247 TC	83.5	5.0	15.5	104.0
		140.5	17.0	26.5	195.0

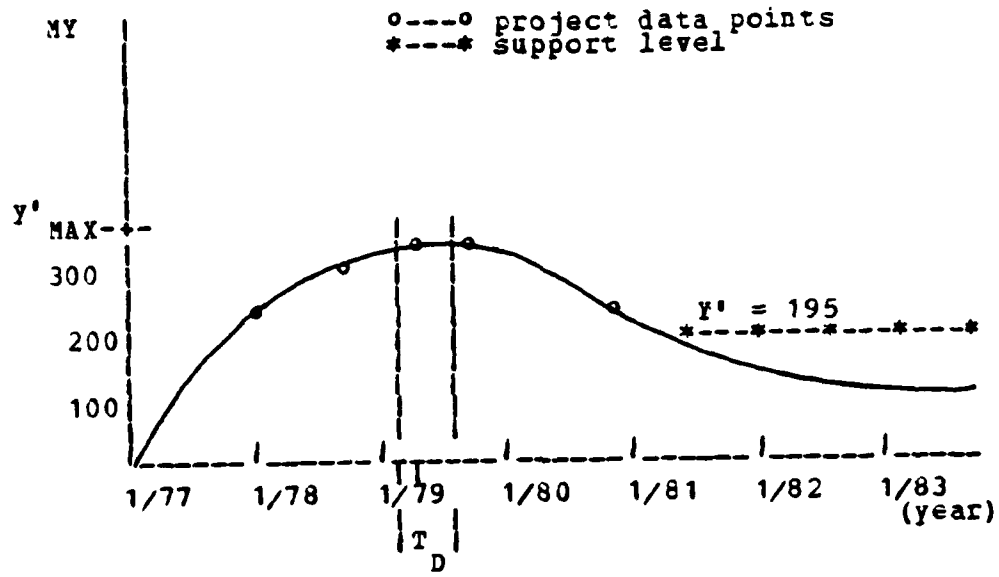
* Matrix Summary represents decomposition of the Space Shuttle Program into major functional areas.

AASD Matrix Estimate Summary *

Area	Code Size	Maint. Level	GPN & Support	M&S	Total
SM	54880	6.7	3.0	1.3	11.0
VCO	57145	5.2	---	.8	6.0
GN&C	129918	16.7	4.0	2.8	23.5
R.A.	------	10.4	5.0	2.6	18.0
P.A.	30975	3.0	---	.5	3.5
AASD M&S	------	---	---	2.0	2.0
Totals	272918	42.0	12.0	10.0	64.0

*Note: This table illustrates an additional decomposition of a major functional area into subcomponents.

Plotted Sample Data



PROJECT CURVE PARAMETERS

$t_d = 2.5$

$k = 1343$

$a = .08$

$y'_{max} = 325$

LIST OF REFERENCES

1. Wegner, P., "Research Directions in Software Technology," Third International Conference on Software Engineering, Proc., pp. 243-259, 10-12 May 1978.

2. Boehm, B., "Software and Its Impact: A Quantitative Assessment", Datamation, v. 19, no. 5, pp. 48-59, May 1973.

3. Baker, F., and Mills, H., "Chief Programmer Teams," IBM Systems Journal, v. 11, no. 1, pp. 56-73, 1972.

4. Boehm, B., "Seven Principles of Software Engineering," Infotech State of the Art Reports: Software Engineering Techniques, pp. 77-113, 1977.

5. Cheatham, Thomas E., The High Costs of Software, National Technical Information Service, 1973.

6. DeRoze, B., and Nyman, T., "The Software Life Cycle - A Management and Technological Challange in the Department of Defense," IEEE Transactions on Software Engineering, SE-4, no. 4, pp. 309-318 July 1978.

7. Kline, N. B., and Schneidewind, N. F., "Life Cycle Comparisons of Hardware and Software Maintainability," Third National Reliability Conference, pp. 4A/3/1-14, 1981.

8. Lehman, J. H., "How Software Projects are Really Managed," Datamation, v. 25, pp. 118-129, 1979.

9. Leintz Bennet P., Swanson, Burton E., and Tompkins, G. E., "Characteristics of Application Software Maintenance," Communications of ACM, v. 21, no. 6, pp. 466-471, 1978.

10. Lyons, Michael J., "Salvaging Your Software Asset (Tools Based Maintenance)," AFIPS, pp. 337-341, 1981.

11. Mills, H. D.,"On the Statistical Validation of Computer Programs," Proceedings of the Second International Conference on Software Engineering, Oct. 13-15, pp.28-37, 1976.

12. Munson, John B, "Software Maintainability: A Practical Concern for Life-cycle Costs," COMPSAC, pp. 54-59, 1978.

13. Reutter, John, III, "Maintenance is a Management Problem and a Programmer's Opportunity," AFIPS, pp. 343-347, 1981.

14. Yau, Stephen S., and Collofello, James S., "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, v. 6, no. 6, pp. 545-552, 1980.

15. Department of Defense Directive 5000.29, Management of Computer Resources in Major Defense Systems, 1976.

16. Bogdan, William R., "Life Cycle Support of Navy Airborne Antisubmarine Warfare Tactical Software", IEEE Computer Software and Applications Conference, Nov 13-16, 1978.

17. Kline, N. B., and Schneidewind, N. F., "Life Cycle Comparisons of Hardware and Software Maintainability," Third National Reliability Conference, pp. 4A/3/1-14, 1981.

18. Leintz Bennet P., Swanson, Burton E., and Tompkins, G. E., "Characteristics of Application Software Maintenance," Communications of ACM, v. 21, no. 6, pp. 466-471, 1978.

19. Lyons, Michael J., "Salvaging Your Software Asset (Tools Based Maintenance)," AFIPS, pp. 337-341, 1981.

20. Munson, John B, "Software Maintainability: A Practical Concern for Life-cycle Costs," COMPSAC, pp. 54-59, 1978.

21. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

22. Reutter, John, III, "Maintenance is a Management Problem and a Programmer's Opportunity," AFIPS, pp. 343-347, 1981.

23. Yau, Stephen S., Collofello, James S., and MacGregor, T. M., "Ripple Effect Analysis of Software Maintenance," Proceedings of COMPSAC, pp. 60-65, 1978.

24. Zelkowitz, M. V., Principles of Software Engineering and Design, Prentice-Hall, 1979.

25. Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, 1981.

26. Boehm, B. W., Brown, J. R., and Lipow, M., "Quantitative Evaluation of Software Quality," Proceeding of the 2nd International Conference on Software Engineering, pp. 592-605, 1976.

27. Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

28. McClure, Carma L., Reducing COBOL Complexity Through Structured Programming, Van Norstrand Reinhold, 1978.

29. Reutter, John, III, "Maintenance is a Management Problem and a Programmer's Opportunity," AFIPS, pp. 343-347, 1981.

30. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

31. Lehman, J. H., "How Software Projects are Really Managed," Datamation, v. 25, pp. 118-129, 1979.

32. Nolan, R. L., "Managing the Crisis in Data Processing," Harvard Business Review, v. 57, pp. 115-126, 1979.

33. DeRoze, B., and Nyman, T., "The Software Life Cycle - A Management and Technological Challange in the Department of Defense," IEEE Transactions on Software Engineering, SE-4, no. 4, pp. 309-318 July 1978.

34. Bogdan, William R., "Life Cycle Support of Navy Airborne Antisubmarine Warfare Tactical Software", IEEE Computer Software and Applications Conference, Nov 13-16, 1978.

35. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

36. Reutter, John, III, "Maintenance is a Management Problem and a Programmer's Opportunity," AFIPS, pp. 343-347, 1981.

37. Baur, H. A., and Birchall, r.h., "Managing Large Scale Software Development With an Automated Change Control System," COMPSAC, 1978.

38. Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

39. Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, 1981.

40. Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

41. Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

42. Chapin, Ned, "Productivity in Software Maintenance," AFIPS, 1981.

43. DeRoze, B., and Nyman, T., "The Software Life Cycle - A Management and Technological Challange in the Department of Defense," IEEE Transactions on Software Engineering, SE-4, no. 4, pp. 309-318 July 1978.

44. Gildersleeve, Thomas R., Successful Data Processing Systems Analysis, Prentice-Hall, 1981.

45. Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, 1981.

46. Leintz Bennet P., Swanson, Burton E., and Tompkins, G. E., "Characteristics of Application Software Maintenance," Communications of ACM, v. 21, no. 6, pp. 466-471, 1978.

47. Munson, John B, "Software Maintainability: A Practical Concern for Life-cycle Costs," COMPSAC, pp. 54-59, 1978.

48. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

49. Yau, Stephen S., and Collofello, James S., "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, v. 6, no. 6, pp. 545-552, 1980.

50. Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

51. Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, 1981.

52. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

53. Rome Air Development Center Report RADC-TR-79-200, Reliability and Maintainability Management Manual, by A. Coppola and A.N. Sukert, pp. 127-151, July 1979.

54. Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (editor), pp. 71-101, Penguin Books, Inc, 1970.

55. Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (editor), pp. 71-101, Penguin Books, Inc, 1970.

56. Boehm, B. W., Brown, J. R., and Lipow, M., "Quantitative Evaluation of Software Quality," Proceeding of the 2nd International Conference on Software Engineering, pp. 592-605, 1976.

57. Thibodeau, Robert and Dodson, E. N., "The Implications of Lifecycle Phase Interrelationships for Software Cost Estimating," Proceedings Second Software Lifecycle Management Workshop, pp. 70-77, 21 August 1978.

58. Lehman, M. M, "Law and Conservation in Large Program Ev.ution ," Proceedings 2nd Software Lifecycle Management Workshop pp. 140-145, 21 August 1978.

59. Belady, L. A. and Lehman, M. M., "A Model of Large Program Development," IBM Systems Journal, v. 15, pp. 225-252, 1976.

60. Lehman, M. M. and Parr, F. N., "Program Evolution and Its Impact on Software Engineering," Proceedings 2nd International Conference on Software Engineering, pp. 350-356, October 1976.

61. Riordon, J. S., "An Evolution Dynamics Model of Software system development," Working Papers of the Software Life Cycle Workshop, August 1977.

62. Parr, F. N., "An Alternative to the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, v. 6, pp. 291-297, 1980.

63. Belady, L. A. and Lehman, M. M., "An Introduction to Growth Dynamics," Statistical Computer Performance Evalution, pp. 503-511, Academic Press, 1972.

64. Lehman, M. M., "On Understanding Laws, Ev.ution, and Conservation in the Large Program Lifecycle," The Journal of Systems and Software, v. 1, pp. 213-223, 1980.

65. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, pp. 67-97, Addison-Wesley, 1980.

66. Rome Air Development Center RADC-TR-77-220, Software Cost Estimation Study: Study Results by Doty Associates, Inc., pp. 15-100, June 1977.

67. Chrysler, E., "Some Basic Determinants of Computer Programming Productivity," Communcations of the ACM, v. 21, pp. 581-587, 1978.

68. Johnson, J. R., "A Working Measure of Productivity," Datamation, v. 23, pp. 106-110, 1977.

69. Walston, C. E. and Felix, C. P., "A Method of Programming Measurement and Estimation," IBM System Journal, v. 16, pp. 54-73, 1977.

70. Walston, C. E. and Felix, C. P., "A Method of Programming Measurement and Estimation," IBM System Journal, v. 16, pp. 54-73, 1977.

71. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, pp. 67-97, Addison-Wesley, 1980.

72. Boehm, B. W., Brown, J. R., and Lipow, M., "Quantitative Evaluation of Software Quality," Proceeding of the 2nd International Conference on Software Engineering, pp. 592-605, 1976.

73. General Electric Command and Information Systems Report GE-TIS-77CI502, Factors in Software Quality by McCall, J. A., Richards, P. K., and Walters, G. F., 1977.

74. Elshoff, J. L., "A Review of Software Measurement Studies at General Motors Research Laboratories," Proceedings of the 2nd annual Software Lifecycle Management Workshop pp. 172-174, 1978.

75. McCabe, T. J. "A Complexity Measure," IEEE Transactions of Software Engineering, v. 2, pp. 308-320, 1976.

76. Rabin, M. O., "Complexity of Computations," Communications of the ACM, v. 20, pp. 625-633, 1977.

77. Gorden, R. D.,"Measuring Improvements in Program Clarity," IEEE Transactions on Software Engineering, v. 5, pp. 79-91, 1979.

78. Halstead, M. H., Elements of Software Science, pp. 46-61, Elsevier North-Holland, 1977.

79. Zislis, Paul, "Semantic Decomposition of Computer Programs: An Aid to Program Testing," ACTA Informatica, v. 4, pp. 245-269, 1975.

80. Halstead, M. H., Elements of Software Science, pp. 46-61, Elsevier North-Holland, 1977.

81. Halstead, M. H., Elements of Software Science, pp. 46-61, Elsevier North-Holland, 1977.

82. McCabe, T. J. "A Complexity Measure," IEEE Transactions of Software Engineering, v. 2, pp. 308-320, 1976.

83. Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements," IEEE Transactions on Software Engineering, v. SE-5, no. 5, pp. 504-514, 1979.

84. Bradley, Gordon H., and others, Structure and Error Detection in Computer Software, Naval Postgraduate School, 1975.

85. Endres, Albert B., "An Analysis of Errors and Their Causes In Systems Programs," IEEE Transactions on Software Engineering, v. SE-1, no. 2, pp. 140-149, June 1975.

86. Goel, Amrit L., "Software Error Detection Model With Applications," The Journal of Systems and Software, v. 1, no. 3, pp. 243-249, 1980.

87. Littlewood, Bev, "Theories of Software Reliability: How Good Are They and How Can They Be Improved," IEEE Transactions on Software Engineering, v. SE-6, no. 5, pp. 489-500, 1980.

88. Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements," IEEE Transactions on Software Engineering, v. SE-5, no. 5, pp. 504-514, 1979.

89. Schneidewind, Norman F., Howard, Gilbert T., and Kirchgaessner, M., Software Error Detection Models, Validation Tests and Program Complexity Measures, Naval Postgraduate School, 1976.

90. Schneidewind, Norman F., Analysis of Error Processes in Computer Software, Naval Postgraduate School, 1974.

91. Taylor Richard N. and Osterweil, Leon J., "Anomaly Detection in Concurrent Software by Static Flow Analysis," IEEE Transactions on Software Engineering, v. SE-6, no. 3, pp. 265-277, 1980.

92. Yau, Stephen S., and Collofello, James S., "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, v. 6, no. 6, pp. 545-552, 1980.

93. Schneidewind, Norman F., Analysis of Error Processes in Computer Software, Naval Postgraduate School, 1974.

94. Littlewood, Bev, "Theories of Software Reliability: How Good Are They and How Can They Be Improved," IEEE Transactions on Software Engineering, v. SE-6, no. 5, pp. 489-500, 1980.

95. Bradley, Gordon H., and others, Structure and Error Detection in Computer Software, Naval Postgraduate School, 1975.

96. Goel, Amrit L., "Software Error Detection Model With Applications," The Journal of Systems and Software, v. 1, no. 3, pp. 243-249, 1980.

97. Littlewood, Bev, "Theories of Software Reliability: How Good Are They and How Can They Be Improved," IEEE Transactions on Software Engineering, v. SE-6, no. 5, pp. 489-500, 1980.

98. Schneidewind, Norman F., Howard, Gilbert T., and Kirchgaessner, M., Software Error Detection Models, Validation Tests and Program Complexity Measures, Naval Postgraduate School, 1976.

99. Schneidewind, Norman F., Analysis of Error Processes in Computer Software, Naval Postgraduate School, 1974.

100. Endres, Albert B., "An Analysis of Errors and Their Causes In Systems Programs," IEEE Transactions on Software Engineering, v. SE-1, no. 2, pp. 140-149, June 1975.

101. Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements," IEEE Transactions on Software Engineering, v. SE-5, no. 5, pp. 504-514, 1979.

102. Motley, R. W. and Brooks, W. D., Statistical Prediction of Programming Errors, Rome Air Development Center, 1977.

103. Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements," IEEE Transactions on Software Engineering, v. SE-5, no. 5, pp. 504-514, 1979.

104. Putnam, Lawrence H., Tutorial - Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers, IEEE, 1980.

105. Putnam, Lawrence H., Tutorial - Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers, IEEE, 1980.

106. Department of the Army DA Pamphlet No. 18-8, A Software Resource Macroestimating Procedure, February 1977.

107. Wonnacott, Thomas H. and Wonnacott, Ronald J., Regression: A Second Course In Statistics, pp. 399-413, Wiley, 1981.

108. Belady, L. A. and Lehman, M. M., "A Model of Large Program Development," IBM Systems Journal, v. 15, pp. 225-252, 1976.

109. Lehman, M. M. and Parr, F. N., "Program Evolution and Its Impact on Software Engineering," Proceedings 2nd International Conference on Software Engineering, pp. 350-356, October 1976.

110. Parr, F. N., "An Alternative tc the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, v. 6, pp. 291-297, 1980.

111. Parr, F. N., "An Alternative to the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, v. 6, pp. 291-297, 1980.

112. Department cf the Army DA Pamphlet No. 18-8, A Software Resource Macroestimating Procedure, February 1977.

113. Rone, Kyle A. Space Shuttle Software Briefing, IBM Federal Systems Division, Huston,Tx., 1980. (unpublished)

114. Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (editor), pp. 71-101, Penguin Books, Inc, 1970.

115. Putnam, Lawrence H., Tutorial - Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers, IEEE, 1980.

116. Rone, Kyle A. Space Shuttle Software Briefing, IBM Federal Systems Division, Huston,Tx., 1980. (unpublished)

117. Belady, L. A. and Lehman, M. M., "An Introduction to Growth Dynamics," Statistical Computer Performance Evalution, pp. 503-511, Academic Press, 1972.

118. Lehman, M. M., "On Understanding Laws, Evolution, and Conservation in the Large Program Lifecycle," The Journal of Systems and Software, v. 1, pp. 213-223, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistics Studies Information Exchange U. S. Army Logistics Management Center Fort Lee, Virginia 23801	2
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 54 Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
5.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Dr. Lyle A. Cox Jr., Code 52CL Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	4
7.	Dr. Danial C. Boger, Code 54BK Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
8.	Dr. Michael G. Sovereign Code 55ZO Department of Operations Research Naval Postgraduate School Monterey, California 93940	1
9.	LCDR Ronald Modes Code 52MF Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
10.	CDR M. L. Sneiderman, SC, USN, Code 54ZZ Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
11.	Dr. H. H. Locmis, Code 62LM Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1

12. Captain Blair R. Vorgang, USMC 1
Headquarters, United States Marine Corp (CCP)
Washington D. C. 20381

13. CDR Sullivan, USN, G60 1
Headquarters, Commander Naval Security Group
3801 Nebraska Avenue
Washington D. C. 20381

14. Mr. Ray Rich G60 1
Headquarters, Commander Naval Security Group
3801 Nebraska Avenue
Washington D. C. 20381

15. Mr. Kyle Y. Rone 1
IBM Federal Systems Division
1322 Space Park Drive
Houston, Texas 77058

16. Mr. William Leary 1
Office of Secretary of Defense for Data Automation
The Pentagon
Washington D. C. 20381

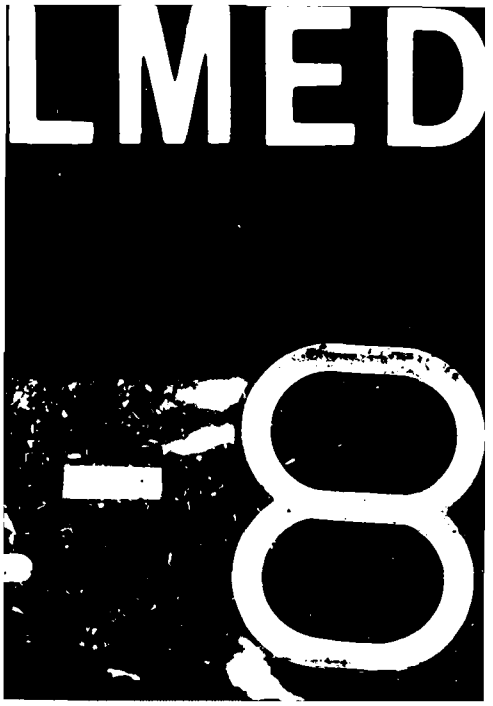
17. Mr. Lawrence H. Putnam 1
Quantitative Software Management, Inc.
1057 Waverly Way
McLean, Virginia, 22101

18. LCDR Daniel. J. Devescovi, CEC, USN 1
U. S. Navy Public Works Center
Box 6
FPO San Francisco, California 96651

19. Dr. Norman F. Schneidewind, Code 54SS 1
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93940

20. LCDR James F. Green 2
Naval Security Group Activity, Box 102
Homestead, Florida 33039

21. LT. Brenda F. Selby 2
Naval Regional Data Automation Center, San Francisco
Alameda, California 94501


LMED
8